




University of  
Stavanger

**Faculty of Science and Technology**

## **MASTER'S THESIS**

Study program/Specialization: Robotic Technology and Signal Processing	Spring semester, 2021.  Open
Writer: Jon Erik Bjerga	 (Writer's signature)
Faculty supervisor: Karl Skretting  External supervisor(s): Torfi Thorhallsson and Martin Eineborg	
Thesis title: Using deep learning to create fake images of previously unseen objects.	
Credits (ECTS): 30	
Key words: Generative Adversarial Network, GAN, DCGAN, Wasserstein GAN, WGAN-GP, Deep Learning, Artificial Neural Network, Robot.	Pages: 62  + supplement material/other: 37  Stavanger, 15. June / 2021.



# USING DEEP LEARNING TO CREATE FAKE IMAGES OF PREVIOUSLY UNSEEN OBJECTS

Master's thesis in Robotic Technology and Signal Processing

BY

**Jon Erik Bjerga**

INTERNAL SUPERVISOR:

**Karl Skretting**

EXTERNAL SUPERVISORS:

**Torfi Thorhallsson and Martin Eineborg**

University of Stavanger  
Faculty of Science and Technology  
Department of Electrical Engineering and Computer Science  
15. June. 2021

# Abstract

Deep learning artificial neural networks are implemented in machines at an increasing rate in order to make them think and act the human way. A popular use of these neural networks is in object detection software, making the machines able to know their environment. A problem is that the training of the neural network requires manual effort, in order to create the training dataset.

Will advances in the training of Generative models, with the introduction of adversarial training be able to reduce this manual effort? This thesis explores if (assesses how) the Generative Adversarial Networks (GAN) model can be used to supplement a training dataset without the manual registration effort.

The main focus is the Deep Convolutional Generative Adversarial Network (DCGAN) and the Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) versions of GANs. The thesis includes the theoretical background and some practical implementation issues for experimenting with these GAN variants.

Three main experiments are presented. First, testing how these generative networks perform on a dataset of images with boxes containing objects with different orientations and positions. Second, experimenting with only a limited amount of images used for training, before testing two proposed improvement methods, pre-training the models and expanding the dataset with the generated (artificial) images. For the last experiment the object detection software will be tested on the generated images.

The results show that it is possible to use Generative Adversarial Networks to generate new additional images for a limited dataset. Perhaps not impressive, but nevertheless, up to 4 % of the generated artificial images have the sufficient quality to be considered as new additions to the dataset.

# Acknowledgements

This thesis concludes my master's degree in "Robotic Technology and Signal Processing" at the University of Stavanger.

I want to give thanks to Karl Skretting for being my primary counselor throughout the semester, giving me valuable guidance, pitching the assignment and introducing me to the company Pickr.AI and their advisors at Reykjavik university.

I also want to thank my colleagues at the university of Reykjavik, Torfi Thorhallsson and Martin Eineborg. For dedicating much valuable time to tutoring meetings, where their inputs and guidance was an invaluable source of information.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Pickr.AI . . . . .	2
1.3	System overview . . . . .	2
1.4	Problem definition (Thesis definition) . . . . .	4
1.5	Thesis outline . . . . .	5
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Artificial Neural Network (ANN) . . . . .	6
2.2	Convolutional Neural Network (CNN) . . . . .	6
2.2.1	Convolutional layers . . . . .	7
2.2.2	Activation layers . . . . .	7
2.2.3	Up- and down-sampling . . . . .	9
2.2.4	Normalization layer . . . . .	10
2.3	Generative models . . . . .	11
2.3.1	Discriminative modeling . . . . .	11
2.3.2	Generative Modeling . . . . .	12
2.3.3	Adversarial feature learning . . . . .	13
2.4	Generative Adversarial Networks (GAN) . . . . .	15
2.5	Wasserstein Generative Adversarial Network (WGAN) . . . . .	16
2.6	Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP) . . . . .	17
2.7	Related work . . . . .	18
2.8	Current approach . . . . .	19
<b>3</b>	<b>Methods</b>	<b>20</b>
3.1	Equipment . . . . .	20
3.1.1	Local computer: . . . . .	20
3.1.2	Google Colaboratory (Colab) . . . . .	21
3.2	Software . . . . .	22
3.2.1	Setting up the Integrated Development Environment (IDE) . . . . .	22
3.2.2	Tensorflow GPU support: . . . . .	23
3.2.3	Virtual environment for the local computer . . . . .	23
3.3	Proposed solution . . . . .	25
3.3.1	Object detector . . . . .	25
3.3.2	GAN (DCGAN) vs. WGAN-GP . . . . .	26
3.3.3	Generate images from a limited dataset . . . . .	26
3.4	Dataset . . . . .	27
3.4.1	Dataset for object detector . . . . .	28

3.4.2	Dataset for GAN generation . . . . .	30
3.5	Object detector YOLOv5 . . . . .	32
3.5.1	Code for YOLOv5 . . . . .	32
3.6	Code for GAN generation . . . . .	33
3.7	Methods of evaluating GANs . . . . .	37
3.7.1	Manual inspection . . . . .	37
3.7.2	Quantitative methods . . . . .	38
3.7.3	Qualitative methods . . . . .	38
3.7.4	Evaluations for this thesis . . . . .	38
<b>4</b>	<b>Results</b>	<b>39</b>
4.1	Experiment: Train and test YOLOv5x . . . . .	39
4.2	Experiment: GAN generated objects . . . . .	41
4.3	Experiment: WGAN-GP generated objects . . . . .	43
4.4	Experiment: WGAN-GP with 10 images . . . . .	46
4.4.1	Experiment: Testing pre-training . . . . .	48
4.4.2	Experiment: Pre-trained WGAN-GP with 10 images . . . . .	49
4.4.3	Experiment: Training with generated images in the dataset . . . . .	51
4.4.4	Experiment: Expanding dataset with 10 images . . . . .	53
4.4.5	Comparing experiment results . . . . .	55
4.5	Experiment: Detect generated objects with YOLOv5x . . . . .	56
<b>5</b>	<b>Discussion and Future Work</b>	<b>57</b>
5.1	Discussion about the experiments . . . . .	57
5.2	Discussion about the suitability . . . . .	60
5.3	Future work . . . . .	61
5.3.1	Deconstructing a full box of objects, with GAN inpainting . . . . .	61
5.3.2	Experimentation's preformed with new GAN versions. . . . .	61
5.3.3	Development of evaluation methods . . . . .	61
<b>6</b>	<b>Conclusion</b>	<b>62</b>

# List of Figures

1.1	The figure displays the conceptualized system for this thesis, which is the extent of this physical systems relevance as it is a theoretical assignment. The thesis is based on the same system, for the same firm as the thesis of Sigurd Ytterstad [1] and his thesis is the source of this figure. On the left is the robot operating the system (area tinted blue), and on the right is the warehouse shelf (area tinted green). On the bottom is a conveyor belt (area tinted red) [1]. . . . .	3
2.1	The x-axis (horizontal) represents the input $x$ and the y-axis (vertical) represents the output function $f(x)$ , figure is gathered from source [2] . . . . .	8
2.2	Illustrative application of strided- and fractional stided convolutions, from source [3]. . . . .	9
2.3	Illustrative examples for the loss function in a neuron with and without batch normalization. The normalization makes it easier for the optimizer to find the steepest slope, because $x$ and $y$ are similar in size. Image sourced from [4], then the labels on axis have been changed. . . . .	10
2.4	The example image shows the GAN process, an input of random noise ( $Z$ ) is passed through multiple repeating layers, creating an fake image. The discriminator passes either the fake or real image class through multiple repeating layers to calculate a prediction. The class and the prediction is used to produce the loss functions. Image source [5], added example parameters to the intervals and changed the notations for the loss functions. . . . .	12
2.5	The color boxes illustrates the parts involved in the back-propagation of the Generator in yellow and Discriminator in red. The process of back-propagation moves from the loss, to the model of same name, with the goal of performing weight adjustments (Generator loss does not affect the Discriminator model). The red line has been added to the image that is borrowed from source [6]. . . . .	13
3.1	Showing the path's needed for this environment. . . . .	23
3.2	Showing the GPU support test output in Pycharm. . . . .	24
3.3	Example images from all the different groups, the groups are referred to with numbers starting at the top left corner (top row is 1, 2 and 3). . . . .	27
3.4	Showing annotated images of the same product images that are going to be used for GAN generation. . . . .	28
3.5	Examples of other images and objects in the dataset. . . . .	28
3.6	Health check performed in the application Roboflow. . . . .	29
3.7	Examples from the two smaller datasets. . . . .	30

3.8	Shows the performance of the YOLOv5 edition 5 models. "COCO AP val" is a value representing the averaged over all categories [7], calculating multiple intersection union between a detected area and an annotated area (validation dataset), this explanation is only valid for the COCO context (elaborated at source [7]). "GPU speed" self-evident as the milliseconds used per image, during training. Image from source [8]. . . . .	32
3.9	Pseudocode for the GAN training algorithm, taken from the original paper [9]. The inner "for"-loop minimizes the function of the discriminator, the outer "for"-loop maximizes the function of the generator, leading to alternation training between the models (one at the time). $k$ is notated as $N_D$ in the code ( $n_{critic}$ in figure 3.10), stating how many times the discriminator will run the training loop for every generator loop (equal to the number of epochs). . . .	34
3.10	Pseudocode for the WGAN-GP training algorithm, taken from the paper [10]. $n_{critic}$ is notated as $N_D$ in the code, stating how many times the critic will run the training loop for every generator loop (equal to the number of epochs). The random number $\epsilon$ is implemented as $\sim N(0, 1)$ not $\sim U(0, 1)$ . The inner "for"-loop calculate the gradient penalty, the outer "for"-loop maximizes the function of the critic and the "while"-loop maximizes the function of the generator. The training alternates between the models (one at the time). Formula for $L^i$ is the same the simplified formula 2.9 in chapter 2.6 . . . . .	35
4.1	Examples from the detection run on the test part of the dataset. . . . .	40
4.2	Examples that show how objects that are visible enough to be annotated are also being detected. . . . .	40
4.3	GAN training process. Showing 5 of 100 sampled images horizontally for every 5000-iteration interval of generator training. Images starting at iteration 250, since the iteration 0 image is just a gray noise image. . . . .	41
4.4	Oscillation during the GAN training process. Showing 1 of the 100 sampled images for iterations between 13 750 to 15 500 during generator training. . .	42
4.5	100 generated images from iteration 39500. A larger version of this image can be found in appendix figure 1. . . . .	42
4.6	Similar versions of one sample image during training, generator iteration listed below the image. A different but very similar image had to be used for iteration 10 000, because there was no similar image sample generated between iteration 9 000 and 16 000 (2800 sample images from 28 sample intervals). . .	44
4.7	100 Generated images from the model. White squares are too bad quality to be new generations and Red squares do already exist in the training dataset. The Yellow squares represent interesting generations and the Green squares that are new acceptable quality generations. . . . .	45
4.8	The reduced dataset of ten images that will be used in this experiment. . . .	46
4.9	The best generated samples from the last iteration is show in figures (a-f), some good generations was observed in earlier samples and shown in figures (g-i). Figure (e) and (h) show a similar new generation, but the quality of the image has decreased with during the longer training. . . . .	47
4.10	Generated samples from training of experiment 4.4 are shown in figures (a-h). Continued training of experiment 4.4 with ten new images as the dataset, are shown in figures (i-p). The sampled intervals are listed below the images. . .	48
4.11	Representative samples of the 100 generated images after 1000 iterations of generator training. . . . .	48



4.12	16 representative samples of the 100 created, to illustrate the model quality after 675 000 iterations. The whole sample image can be viewed in appendix figure 7. . . . .	49
4.13	The best generated samples from the last iteration. . . . .	50
4.14	The best generated samples from the last iteration. . . . .	51
4.15	The best generated samples during training of the network. The sample iteration is listed below the images. . . . .	52
4.16	The additional ten images that will be added for this experiment. . . . .	53
4.17	The best newly generated samples from the last iteration. . . . .	54
4.18	The best generated samples found during training in experiment 4.3. . . . .	56
4.19	The best generated samples found during training in experiment 4.4. . . . .	56
4.20	The best generated samples found during training in experiment 4.4.3. . . . .	56
1	GAN trained with 151 images for 39 500 iterations. . . . .	
2	WGAN-GP trained with 151 images for 20 000 iterations. . . . .	
3	WGAN-GP trained with 151 images for 30 000 iterations. . . . .	
4	100 generated samples after 8 000 iterations of training on 10 images . . . . .	
5	100 generated samples after 9 000 iterations of training on 10 images . . . . .	
6	100 generated samples after 10 000 iterations of training on 10 images . . . . .	
7	Pre-trained model after 675 000 iterations of training on 1 801 images . . . . .	
8	100 generated samples after 8 000 (683 000) iterations of continued training with a pre-trained model on 10 images . . . . .	
9	100 generated samples after 9 000 (684 000) iterations of continued training with a pre-trained model on 10 images . . . . .	
10	100 generated samples after 10 000 (685 000) iterations of continued training with a pre-trained model on 10 images . . . . .	
11	100 generated samples after 8 000 (693 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images . . . . .	
12	100 generated samples after 9 000 (694 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images . . . . .	
13	100 generated samples after 10 000 (695 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images . . . . .	
14	100 generated samples after 18 000 iterations of training with 20 real images	
15	100 generated samples after 19 000 iterations of training with 20 real images	
16	100 generated samples after 20 000 iterations of training with 20 real images	

# Abbreviations

**AI** Artificial intelligence

**ANN** Artificial Neural Network

**CNN** Convolutional Neural Network

**CPU** Central Processing Unit

**DCGAN** Deep Convolutional Generative Adversarial Networks

**GAN** Generative Adversarial Networks

**GPU** Graphics Processing Unit

**GP** Gradient Penalty

**IDE** Integrated Development Environment

**RAM** Random Access Memory

**RGB** Red, Green and Blue

**TBU** Tensor Processing Unit

**WGAN** Wasserstein Generative Adversarial Network

**YOLO** You Only Look Once

# 1. Introduction

## 1.1 Motivation

Technology is rapidly advancing in the field of artificial intelligence (AI). This technology adjusts (or learn) large sets of parameters in complex algorithms to make machines mimic the human way of thinking and acting [11]. These types of machines have become ever more applicable to handle a wide range of tasks. Creating an interest in utilizing them for industrialized AI automation of physical tasks.

This type of automation will initially be most suitable for simple, repetitive and time-consuming tasks. An industry that incorporate many tasks of this type are online stores. Specifically, the logistics performed on behalf of sales.

The logistics for order fulfillment mainly consists of the picking and packing of customer orders. A well-developed AI automation system may have the possibility of limiting the human labor required for this process, with the AI helping the system handle tasks more autonomously from humans. Assumably, leading to cost reductions, increased efficiency and quality.

If an automation investment is successful, leading to all aforementioned benefits, an online store will be able to focus more on their core business. Creating a potential for improved performance in other areas as well.

In the last years from 2016 to 2020, electronic commerce has had an 87.7% increased turnover [12]. This statistic includes the industry of online stores. Norway's largest online beauty shop is "Bli Vakker", established in 2007, the company has reached almost 1 000 000 registered customers [13]. A growth that probably has inspired the interest in optimization through automatization of the picking and packing processes. For this purpose, Bli Vakker has hired the help of the automation specialized company "Pickr.AI".

## 1.2 Pickr.AI

Pickr.AI is an automation company, that specializes on the automation of the logistical operations involved in order fulfillment. The company focuses on methods for cutting cost across this entire value chain [14]. Trying to achieve this with innovative solutions, the current aim is to automate entire warehouses. The company is in a startup/development state, with varying challenges, customer demands and orders.

One ongoing assignment at Pickr.AI, is the robotical automatization of order fulfillment in online stores [1]. The goal is limiting human interaction on the process, thereby making it more cost-effective with a higher return yield.

Pickr.AI is thereby interested in new technology and innovative methods, that may bring improvements for their development of an automatic robotic order fulfillment station. Especially development in the sub-field of AI, called Deep Learning, enabling the machines to learn without human supervision [15]. They hope that implementing Deep Learning processes will reduce the manual registration effort associated with introducing new objects for picking and packing.

This master's thesis was proposed by advisors of Pickr.AI at Reykjavik university, inspired by the progress of research within "Deep Learning" and "Generative Adversarial Network".

## 1.3 System overview

The assignment for the thesis is a theoretical one, but the solution is conceptualized to be used on a fully automatic robotic order fulfillment station. This stations system consists of a robot arm able to move and handle objects in the X, Y and Z axes. It is placed along a conveyor belt, as viewed in figure 1.1. Boxes containing one or more objects will arrive along the conveyor belt, ready for picking. The purpose of the system is to be able to recognize objects that arrive in the boxes, determine how to handle the objects and then transport the objects to a new location.

The robot is vision guided by a RGB camera and a depth sensor. It receives data about the position of itself, the belt and are able to find the boxes containing the objects. From there it is able to place the camera to have a full view of the box and the content. The control system for the station is an artificial neural network that can be trained with data to handle the objects. Therefore, it will with a sufficient amount of training data containing images and annotations, be able to pick up objects and place them at a new location.

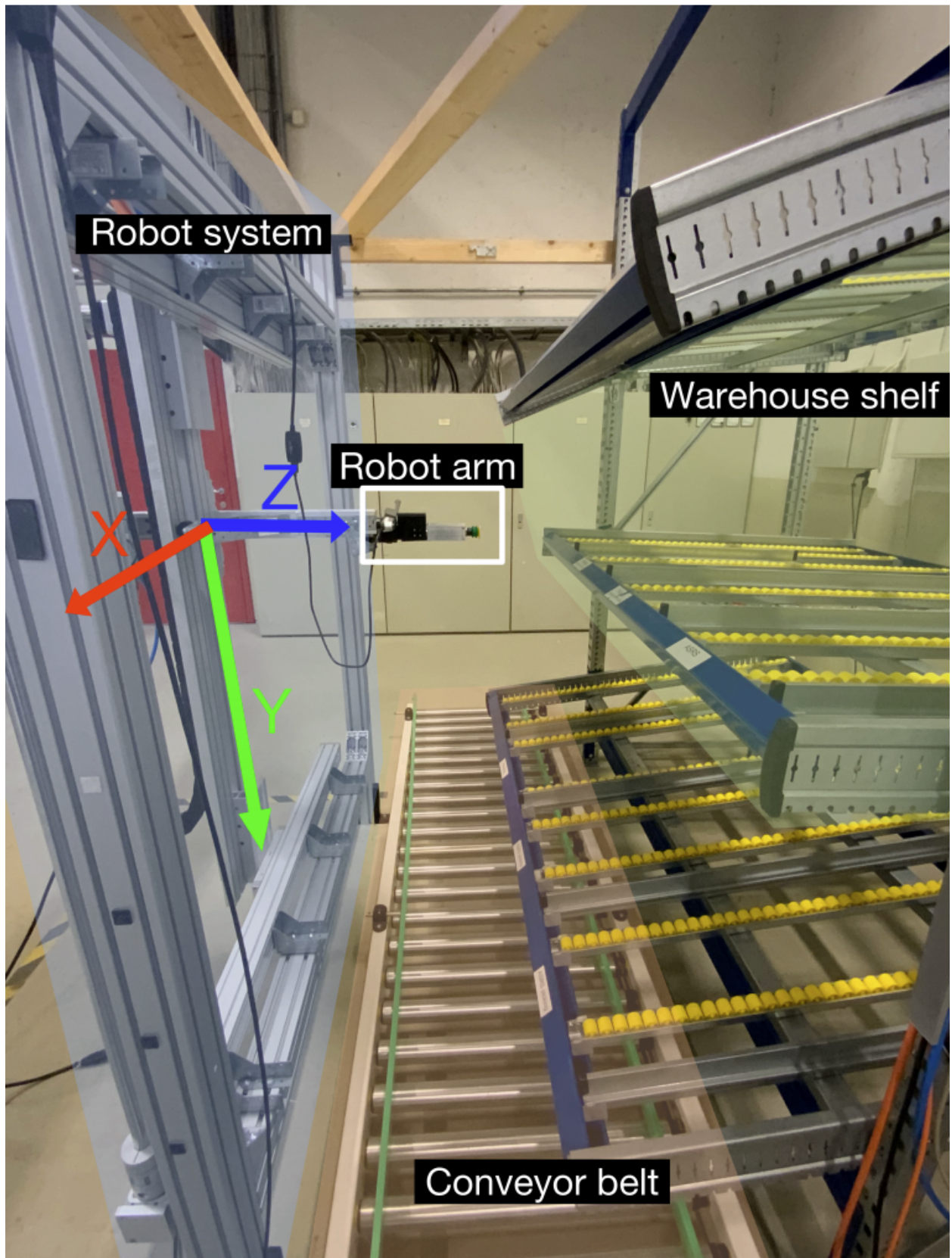


Figure 1.1: The figure displays the conceptualized system for this thesis, which is the extent of this physical systems relevance as it is a theoretical assignment. The thesis is based on the same system, for the same firm as the thesis of Sigurd Ytterstad [1] and his thesis is the source of this figure. On the left is the robot operating the system (area tinted blue), and on the right is the warehouse shelf (area tinted green). On the bottom is a conveyor belt (area tinted red) [1].

## 1.4 Problem definition (Thesis definition)

The problem definition proposed for this thesis is a reformulated version of the problem defined by Pickr.AI. It was changed during the initial tutoring meetings with the advisors of Pickr.AI at Reykjavik university, based on them recommending to explore "Generative Adversarial Networks", providing more information about the problem and supplying a dataset.

The main change is that this exploration requires at least a limited amount of images, not just a single image, as proposed by Pickr.AI. Subsequently a part about determining suitable methods was replaced by determine if Generative Adversarial Networks may provide a suitable method. Additionally, a need for a dataset from an actual fulfillment station, was acquired though the advisors from a earlier master thesis [16] written on request from Pickr.AI. With an additional amount of information added, the reformulated problem definition is as follows:

*Pickr.AI is developing a fully automatic robotic order fulfillment station. The station uses a vision guided robot to pick and pack objects. To achieve this task the system uses an artificial neural network that is trained to handle the objects. The network requires a large labeled image-dataset to train the model for object handling.*

*Creating and quality controlling the dataset is a very time-consuming manual interaction on the system. There is thereby an interest in exploring methods for reducing the manual registration effort associated with introducing new objects for picking, as the picking process occur before the packing process.*

*Given a limited amount of images containing a box filled with previously unseen objects, the objective is to automatically generate new artificial images containing different arrangements and perspectives of the objects. The set of generated images may then be used to train the network, to handle the new objects.*

*Both the image generation and the object detection would be done using methods of deep learning. A part of the problem is to determine if Generative Adversarial Networks may provide a suitable method. This process is intended to reduce the manual registration effort associated with introducing new objects for picking.*

- *For training and validation purposes, real images from an actual fulfillment station, will be substituted with the dataset "RU Beiersdorf" created by Sverrir Bjarnarson for his master thesis [16].*
- *Programming will be primarily conducted in Python using open source computer vision and machine learning frameworks. Network training will need to be done on a GPU server.*

## 1.5 Thesis outline

### **Chapter 2 - Background:**

The chapter provide the theoretical background of the proposed method, additionally it will inform about related work and the current approach.

### **Chapter 3 - Methods:**

The chapter covers the proposed solutions and the preparations required for testing the proposed solution in chapter 4, including equipment and software setup, datasets, code implementation and evaluations.

### **Chapter 4 - Results:**

This chapter will present the experiments conducted and the corresponding result are presented.

### **Chapter 5 - Discussion and Future Work:**

This chapter discusses the results from the experiments preformed in chapter 4, it will also include suggestions for future work.

### **Chapter 6 - Conclusion:**

This chapter covers the conclusion to this thesis.

## 2. Background

This chapter introduces the background to give a fundamental understanding of the methods in chapter 3, including related work and the current approach.

Deep Learning is one type of artificial neural networks, it refers to the number of layers that compose the neural network. The network is required to have at least three layers to be a deep neural network. All the neural networks in this thesis will be of the type deep neural networks.

### 2.1 Artificial Neural Network (ANN)

Artificial neural network can be used for a wide arrangement of tasks and refer to the algorithm that an ANN perform. Every ANN is built up of multiple layers, each of these layers contain multiple neurons[17]. Just as the brain it is modeled after, each of these neurons take multiple inputs and calculate an output based on the weights (neuron-weights) associated to the inputs, imitating the process of resonating towards a decision.

An ANN can be simplified into having three categorical parts, an Input layer, Hidden layer and Output layer. The input layer is for the insertions of data and the output layer is for end prediction based on neuron activations throughout the network. It is in the hidden layers of the neural network all the mathematics and featuristics that make an ANN unique occur. The neurons that make up the layers perform a linear transformation, passing it on to an activation function that determine if the neuron should activate. This process from input to output layer is called a forward-propagation[18]. The process of training the network by updating the weights and biases is called back-propagation[18], where the update is determined by an neurons error passed through a loss function. Determining the weights in an artificial neural network is a result of training and learning to feature extract information from given data.

### 2.2 Convolutional Neural Network (CNN)

As ANN and other pattern detection methods have been around for many decades, one type has become very dominant in the recent years "Convolutional Neural Networks", being particularly true for pattern detection in images. The advantages are summarized as to help the network in obtaining ruggedness to shifts and distortions effecting the images without having fully connected layers. When the advantages are compared to an ANN with fully connected layers, the CNN require less computational power, has fewer parameters and therefor gives an improved performance [19]. A modern CNN are usually comprised of convolutional, pooling, activation and normalization layers. The normalization layer is not a requirement for the creation, but an improvement addition. In the following sub-chapters is an explanation of the building blocks required for a CNN, that are to be used in a GAN.



## 2.2.1 Convolutional layers

As indicated by the name, the convolutional layer is the main feature of a CNN. The layer has multiple parameters consisting of learn-able filters (kernels). Each filter is repeatedly applied to input-results to create a feature map, a map of activation's that indicate the location and strength of detected features from the input [20].

This input region that produces a feature is of a fixed square size called a receptive field. The receptive field consists of pixel values for input layers, and feature maps from previous layers in the following sequential layers. The movement of the application of the filter relative to the input image is referred to as the stride [21]. Movement of the stride is almost always symmetrical in the width and height dimension, and will affect how the filter is applied and the size of the resulting feature map.

The last component of a convolutional layer is something called zero padding, a technique for inventing mock inputs for non-existing inputs [22]. Since it is required that the size of the previous layer divided by the size of the receptive field and the stride result in an integer. The zero padding can be used if the values are not cleanly divisible, so that the receptive field does not attempt to read of the edge of the input, instead reading the mock input.

Convolutional layers exist for both convolutions and transpose-convolutional operations (fractional stride is used for transpose-convolutional, elaborated in chapter 2.2.3). The difference is that a transpose-convolutional operation is designed for up-sampling, implementing an even spread of predefined zero pixels into the input, before performing the convolutional operation with the filter [23]. The operation is performed with feature maps as inputs and the filters weights are learn-able exactly similar to the normal convolutional operation [24]. The convolutional layers can be applied as 1, 2 or 3 dimensional layers, most commonly it is used as a 2-dimensional layer for feature extraction in images.

## 2.2.2 Activation layers

As the neural network is modeled on the human brain, the activation function is modeled to replicate the response from one brain neuron activating. As a neuron takes a weighted sum of inputs and a bias, then it linear transform the information and passes it unto an activation layer that determine if the output is relevant for prediction or if it should be ignored [25]. This is done by performing a transformation (often non-linear) plotting the weighted sum as a normalized output within a specific range.

There are many different activation functions and they are used for different cases, exclusively or coexistent within a neural network. Determining the most suitable function can make the network converge easier and faster. There are no concrete answer when determining the correct function to a specific situation, but there are a rule of thumb, starting with the ReLU function for all hidden layers and ReLU are only supposed to be used in the hidden layers [25]. Then applying another function if ReLU proves sub-optimal.

GAN generation almost exclusively uses the following activation functions, that are well documented at sources [26] and [27], summarized below:

### Sigmoids:

This activation function can be used for all real value inputs and will output the values in the range from 0 to 1 [27]. The function is calculated with  $f(x) = \frac{1}{1+e^{-x}}$ , with  $\mathbf{x}$  as input and  $e$  as the natural logarithm (therefor it is also called the logistic function). Illustrated in figure 2.1 as (a).

### Hyperbolic Tangent Activation Function (Tanh):

This function is very similar to the Sigmoids, the difference is that it outputs the values in the range -1 to 1 [27]. It does this with the function  $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ . Illustrated in figure 2.1 as (b).

### Rectified Linear unit (ReLU):

The ReLU activation function can take all values, it returns 0 for all negative values and the value itself for positive values [28]. The function is as follows  $f(x) = \max(0, x)$ , where **max** is a function for finding the maximum value. Illustrated in figure 2.1 as (c).

### LeakyReLU:

The LeakyReLU is based on the ReLU activation, the difference is that it returns scaled values for negative input values [26]. Function for LeakyReLU is  $f(x) = \max(\alpha \cdot x, x)$ , where  $\alpha$  is a scaling factor often around the value of 0.01. Illustrated in figure 2.1 as (d).

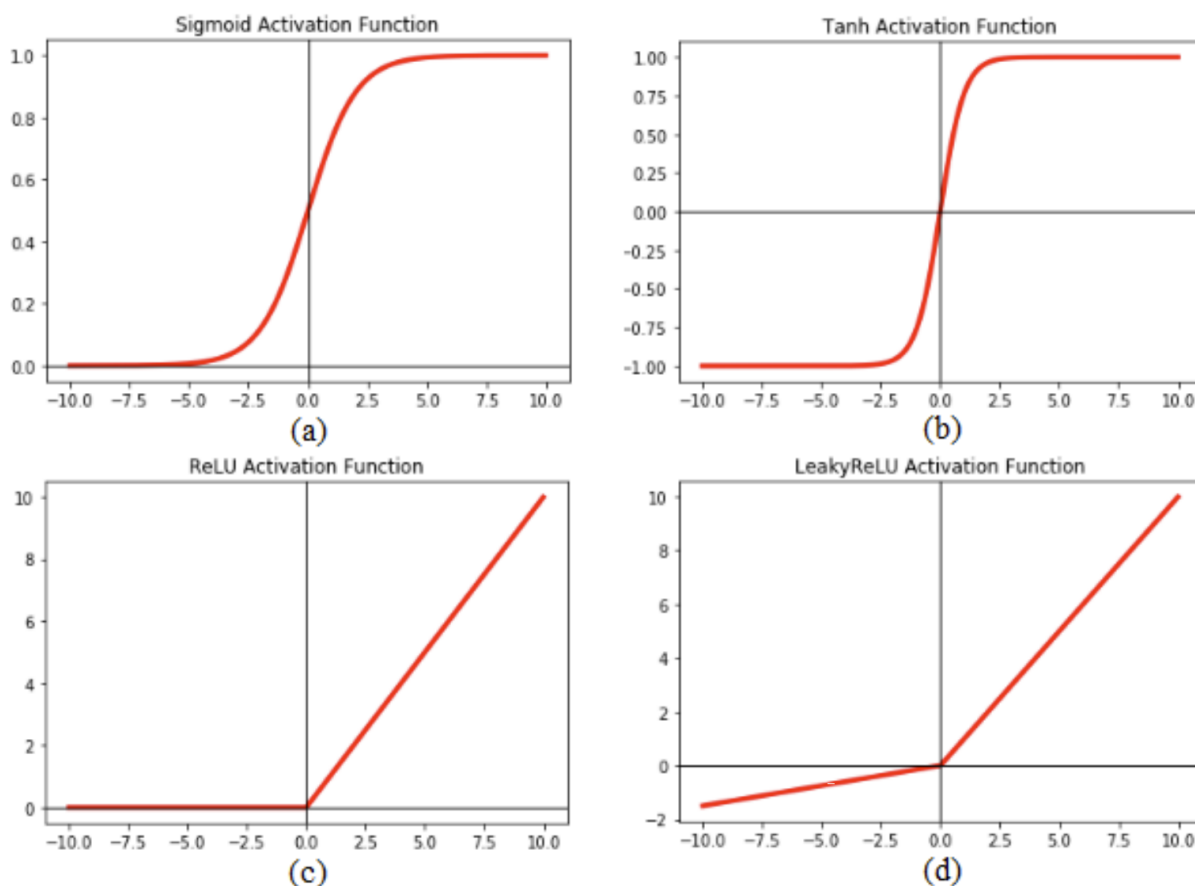


Figure 2.1: The x-axis (horizontal) represents the input  $x$  and the y-axis (vertical) represents the output function  $f(x)$ , figure is gathered from source [2]

### 2.2.3 Up- and down-sampling

Feature maps have from the output of convolutional layers recorded a precise position of the features, this means that it risks creating a different feature map if there are a small position change in a feature [29]. These small movements may occur as a result of rotation, shifting, re-cropping etc. Therefore down-sampling is used as an effective tool, creating a version of the input with lower resolution, reducing the details in the image while containing the larger structural elements.

The most common approach is using pooling layers (with max pooling[29]) after the activation layer to achieve down-sampling, but for GANs it is recommended with methods that utilize the stride (mentioned in chapter 2.2.1) for this operation [30] [31]. More specifically it is recommended to use strided convolutions for down-sampling in the discriminator and fractional-strided convolutions in the generator. Using the stride allows for the CNN to learn its spatial sampling and adjust the spatial resolution fittingly [3].

Strided convolution (**floor** denotes the function rounding down to nearest integer):

$$spatialresolution = \left\lfloor \frac{input_{size} + 2 \cdot \mathbf{floor}(padding)}{stride} \right\rfloor + 1 \quad (2.1)$$

Fractional strided convolutions:

$$spatialresolution = stride \cdot (input_{size} - 1) + kernel \quad (2.2)$$

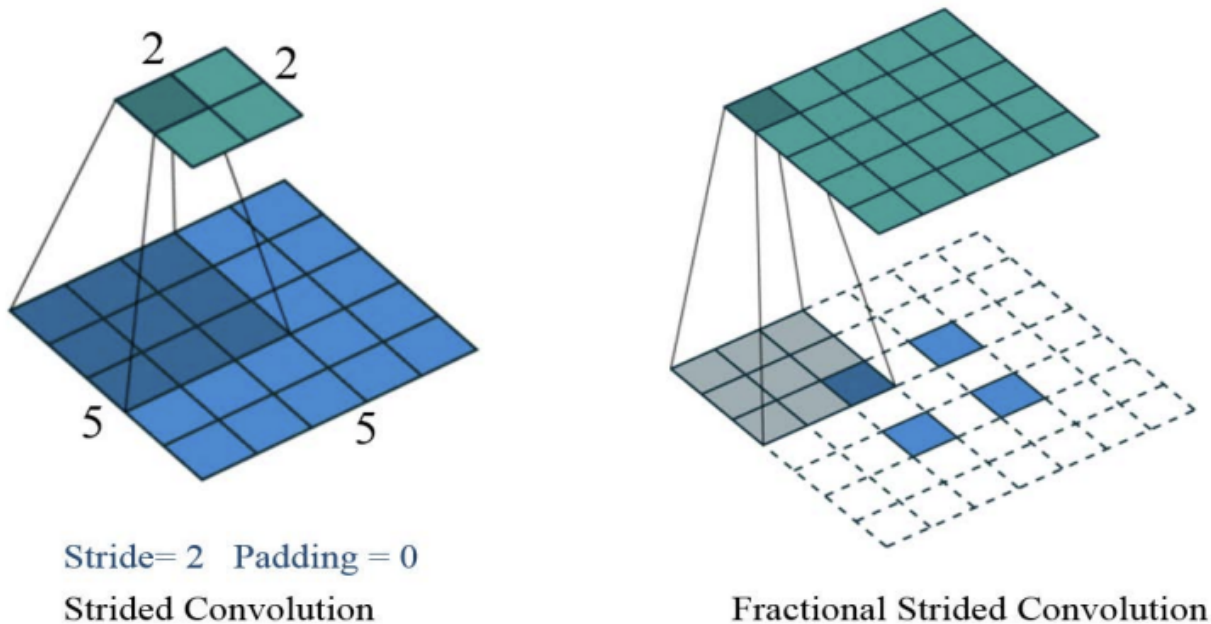


Figure 2.2: Illustrative application of strided- and fractional strided convolutions, from source [3].

The sampling is implemented in the convolutional layer, where the key factor between a normal sampling (or scaling) and the convolutional-sampling is the implementing of filters with learn-able weights [24][23].

## 2.2.4 Normalization layer

Normalization layers are not a required part of CNN, but an improvement added to deep neural networks for stabilizing during the training process. The layers can be applied both before and after activation layer, resulting in some differentiation. The most common way of using them are before, this also apply for GAN generation. Normalizing is a pre-processing tool used for normalizing, scaling and shifting data, without causing distortion. There are many methods for normalizing, for GAN the most common is "Batch Normalization" (BN).[32][4]

Batch Normalization is the normalization with the use of activation values ( $x_i$ ) from a determinant number of samples, called a mini-batch ( $i_{size}$ ). From this the mean ( $\mu$ ) and variance ( $\sigma^2$ ) of the mini-batch is calculated. Then each of the values are normalized with the formula  $\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$ , where  $\epsilon$  is a noise parameter to ensure that the denominator  $\neq$  zero. The result is a transformed activation, that has become standardized, having a mean of zero and unit standard deviation. Because the mean and variance is very reliant on the mini-batch, two new train-able parameters are introduced for each neuron, called  $\gamma$  and  $\beta$ . These new parameters are used to calibrate the mean and the variance, by training over multiple mini-batches the values will converge towards a true mean for  $\gamma$  (re-scaling) and the true variance for  $\beta$  (shifting). The output from the neuron is therefore defined by the function  $y_i = \gamma \cdot \hat{x}_i + \beta$ . [32][4][33][34][35][36][37]

The reason for using the normalization can be visualized by the figure 2.3 below, where the differentiation show that as the activation values have become more symmetric with the transformation into a standard normal distribution with  $N(\mu = 0, \sigma^2 = 1)$ . The transformation has smoothed-out and more evenly distributed the activation value range. Allowing for the use of a larger learning rate as an optimizer is able converge more directly, than previously (because it is less affected by oscillations), reducing the training time. Additionally, it has removed the impact of sub-optimal initial values, as the loss function will converge towards a minimal or a maximum in a similar amount of time, regardless of initial value.[32][4][33][34][35][36][37]

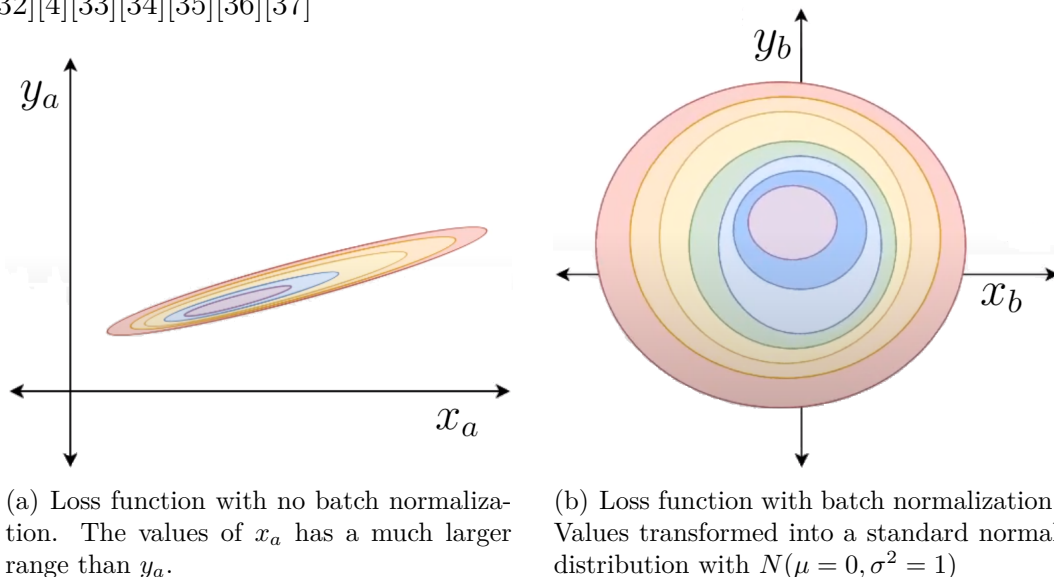


Figure 2.3: Illustrative examples for the loss function in a neuron with and without batch normalization. The normalization makes it easier for the optimizer to find the steepest slope, because  $x$  and  $y$  are similar in size. Image sourced from [4], then the labels on axis have been changed.

There are still other GAN-versions like WGAN-GP that utilize "Layer Normalization" (LN). The LN-method is inspired by BN but created to overcome the dependency of mini-batch [38]. It was proposed by Geoffrey Hinton, to normalize the activation along the feature direction instead of mini-batch direction. This method is used for the discriminator in a WGAN-GP.

## 2.3 Generative models

Generative models are designed to create specific data, based on learned patterns and distributions found in a dataset. They are used for predictions, supplementing, recreating, etc. It is the ability to generate something with an aspect of randomness, that gives the models the unique usefulness. Suggesting the next word in a sentence, based on a dataset, is a prediction. Creating more images of faces, will supplement a dataset with more data to draw from. Recreating a normally existing feature of an image, will recreate a complete image.

GAN is a result of research into Generative models and belong to this subgroup of deep neural network models. It is used as an option for training a generative model. The distinctive feature for a GAN stem from the adversarial-structured learning process, that in turn creates a framework that allows for normally supervised learning approaches of neural networks in an unsupervised way [39].

This means that a discriminative model is incorporated into the system as an adversary to the generative, then the adversarial-structured learning process is built up around the models. It has been discovered that models containing convolutional and convolutional-transpose layers give a greater predictive power to the two models. This type is called "Deep Convolutional Generative Adversarial Network" (DCGAN) and has become synonymous as a normal GAN, as there have been a broad adaptation of these layers in almost all GANs. Therefore, DCGAN will be the focus in this thesis, hereafter referred to as GAN.

### 2.3.1 Discriminative modeling

The discriminative model in a GAN, serves the main purpose of training a generative model. The way this works is by building and training a complete discriminative convolutional neural network. Any general architecture for a CNN based classifier with one real class at the output can be used to design the discriminator [40]. The goal of the discriminator is to determine if an image is real or fake, therefore it only needs to output an 1 or 0. To calculate and determine an output, the image is pre-processed in an input layer that transform the data to artificial neurons that can be further processed in subsequent layers [41]. The subsequent layers consist of a repeating layer group as viewed on next page (Discriminator model layout), not having the normalization in the first group as that leads to sample oscillation and model instability according to [31]. The repeating layers feature extract and down-sample the image by half every rotation, until it has a 4x4 pixel shape. Then it is passed through a single convolutional layer, for a final output.

## 2.3.2 Generative Modeling

The generative model is designed to create new data instances recognizable as a constitute of the dataset, from a given input. The input for a traditional GAN is random noise, there are versions that uses a non-random input, but they serve another purpose than the one sought after in this thesis. The reason for using a randomized input is to ensure that the model does not generate the same output every time [42][6].

What is called random noise, is a latent space of a  $X$ -dimensional (common to use  $X=100$  or  $X=\text{batch size}$ ) hypersphere consisting of variables drawn from a normal distribution ( $\sim N(0,1)$ ) [43]. From the input layer the random noise is projected and reshaped, before passing to repeating hidden layers. These layers will feature extract, include learn-able weights, create feature maps and use fractional-strided transpose convolutions to up-sample the input. These layers will be repeated until the image is of desired pixel and channel size.

### Discriminator model layout:

1. Input layer
2. Repeating layers
  - (a) Convolutional 2D layer
    - with down-sampling
  - (b) Normalization layer (from second repetition)
  - (c) Activation layer
3. Output layer
  - (a) Convolutional 2D layer

### Generator model layout

1. Input layer
2. Repeating layers
  - (a) Convolutional transpose 2D layer
    - with up-sampling
  - (b) Normalization layer
  - (c) Activation layer
3. Output layer
  - (a) Convolutional transpose 2D layer
  - (b) Activation layer

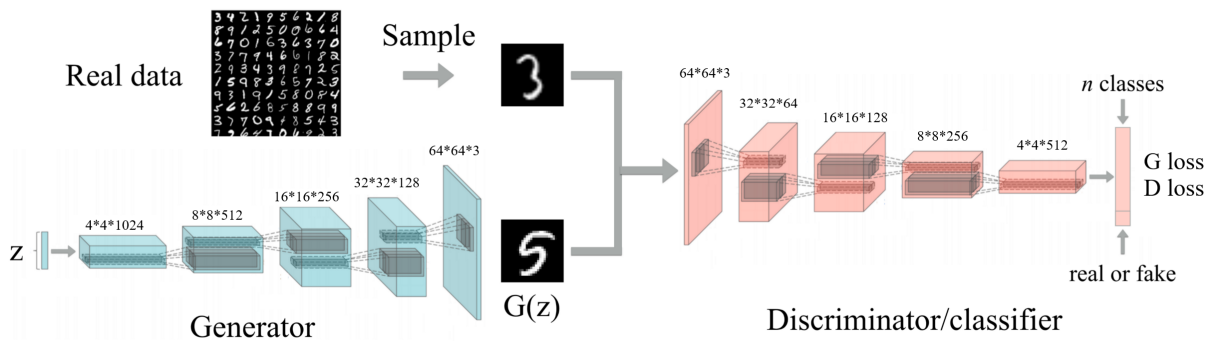


Figure 2.4: The example image shows the GAN process, an input of random noise ( $Z$ ) is passed through multiple repeating layers, creating an fake image. The discriminator passes either the fake or real image class through multiple repeating layers to calculate a prediction. The class and the prediction is used to produce the loss functions. Image source [5], added example parameters to the intervals and changed the notations for the loss functions.

### 2.3.3 Adversarial feature learning

In the training process of a GAN there are two different networks that need to be trained. The training process uses a dataset consisting of real images as real data and generated images as fake data, then creating both a discriminator loss and a generator loss based on the classification success in the discriminator [44]. These losses are afterwards used to penalize the models, updating the weights with the use of back-propagation [45][6]. To create stable losses for training the two models, they cannot be trained simultaneously [46]. That would result in fluctuations in the fake data. Thereby creating fluctuations in the weights of the discriminator, increasing the risk of the generator never converging.

Introducing an alternating training process, ensures constant training parameters during training. This process starts by training the discriminator with back-propagation from the discriminator loss, for one or more epochs. Then training the generator with back-propagation from the generator loss, for one or more epochs. Repeating the process until a convergence can be identified. Note that the discriminator creates the generator loss, the loss does not have impact the discriminator training and is ignored by the model afterwards.

When the GAN converges as a result of improvement to the generator, it directly contributes to a decrease in the discriminators ability to classify correctly. Arriving at a 50 % confidence level for the discriminator, means that the generated images are sufficient in fooling the classifier. Continued training of the discriminator will then be done on feedback of little value, increasing the risk of a collapse in model quality.

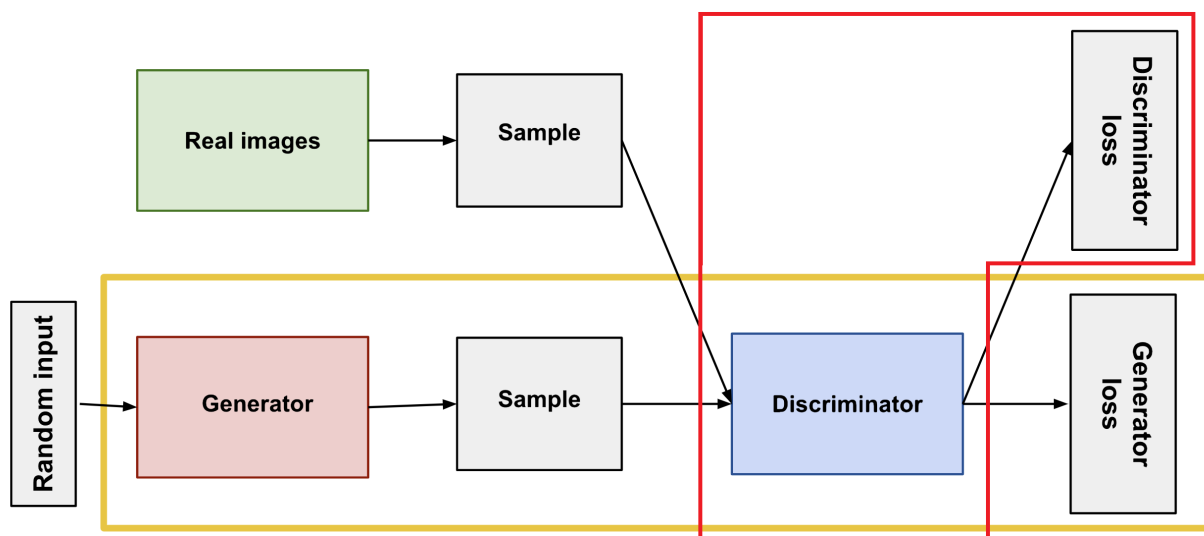


Figure 2.5: The color boxes illustrates the parts involved in the back-propagation of the Generator in yellow and Discriminator in red. The process of back-propagation moves from the loss, to the model of same name, with the goal of performing weight adjustments (Generator loss does not affect the Discriminator model). The red line has been added to the image that is borrowed from source [6].

### Loss functions and optimizers:

The main difference between versions of GAN are commonly connected to experimentation with different loss functions and how they are applied. The objective for the loss function is to evaluate attempts made by the neural network, by calculating the error of a given set of weights, faithfully distilling all the weights down to a single scalar [47]. The goal of training is to minimize or maximize the loss function, by utilizing an optimizer to navigate the slope/gradient of the chosen loss function.

There are experimentation with different optimizers, but there are created a precedence that "Adaptive Moment Estimation" (ADAM) is the best general optimizer, since it achieves good results fast, it has become widely adopted within the field of deep learning neural networks [48]. ADAM implements advantages from two other popular optimizers "AdaGrad" and "RMSprop", seemingly replacing them in most applications [49]. ADAM is a gradient based optimizer algorithm for updating the weights in a neural network, based on adaptive estimates of lower-order moments [50]. These estimations are calculated by the exponential moving average of the gradient with the squared gradient and separate parameters, which control the decay rates [51]. The original revised paper on ADAM can be found at [50], the inner workings of ADAM are well summarized and further explained by Jason Brownlee at [48].

There are multiple sources stating their recommendation of ADAM. The book [52] writes that is the best practice when training DCGAN, same author has listed it as tip nr. 5 at source [30] for training a stable GAN. Furthermore, it is listed as improvement tip at [53] that the ADAM optimizer usually works better than other methods. The optimizer "Stochastic gradient descent" (SGD) is shown (in table 1 and Fig.4) to perform slightly faster than ADAM for original GAN in the paper [51] that is focused batch size comparisons. A comparison at [54] (between SGD, RMSprop and ADAM), conclude that ADAM performs best. Additionally, a deeper research and test into why ADAM is so successful and widely used is performed at [55], where the paper focuses mainly on the advantages that ADAM has over RMSprop and SGD.

As the GAN consists of two neural networks, it will also have two loss functions that may use the same function. These functions reflect the probability distribution of data, one function for the generated data and one for the real data [56][57]. How the two functions work together to give a reflection of the distance measure between the probability functions, is an active area of research. As a result, there have been created multiple GAN versions, where the adversarial learning process makes them hard to evaluate against another. There is no loss function created by the generator, as the model rely on indirectly being trained by the discriminator. In a normal neural network, like the discriminator, the function can be used to objectively assess the training process and model quality. In a GAN the object is to train to obtain an equilibrium between the models and then maintaining it. There have been more research into the development of GAN than the evaluation of them, still some research has been performed in that regards as well, something further elaborated in chapter 3.7.



## 2.4 Generative Adversarial Networks (GAN)

GAN was first described in the 2014 paper "Generative Adversarial Nets" [9], by its inventor Ian Goodfellow and multiple contributors. The idea was to create a framework for teaching deep learning models to capture the probability distribution from a dataset, then becoming able to generate completely new data suitable to be constituents. The framework is created by two distinct models called a generator and a discriminator. The generator is designed to creating artificial (fake) images, similar to the training data, from random noise. While the discriminator is designed to determine if a image is an original (real) or a fake image.

The models are trained by trial and error, pitting themselves against each other as adversaries, as described in chapter 2.3.3. This process continues until the generator is able to generate images of a quality that puts the discriminator at an 50% confidence level in determining the realism of an image [58][59]. This is when the process has arrived at an equilibrium, where the opposite models have completed training each other for mutual gain. As the training is completed, the two models can be saved and utilized for the desired purpose, normally only one of the models are desired and that is usually the generator.

The GAN (DCGAN) is implemented by the minimax loss function, where two functions (written below) calculate the losses based on the probability of the discriminator output passed through a Sigmoid function [60]. The calculations of loss is derived from binary cross-entropy, as a quantification of the difference between real and fake probability distributions [57].

**Discriminator:**

$$\nabla_{\theta_d} \cdot \frac{1}{m} \cdot \sum_{i=1}^m [\log(D(x^{(i)})) + \log(1 - D(G(z^{(i)})))] \quad (2.3)$$

**Generator:**

$$- \nabla_{\theta_g} \cdot \frac{1}{m} \cdot \sum_{i=1}^m [\log(1 - D(G(z^{(i)})))] \quad (2.4)$$

- $x$  = real image
- $z$  = random input ( $\sim N(0,1)$ )
- $D$  = discriminator estimate of the probability that the input instance ( $x$  or  $G(z)$ ) is real.
- $G$  = generated output given an input ( $z$ ).
- $m$  = batch size
- $i$  = index

## 2.5 Wasserstein Generative Adversarial Network (WGAN)

The training of GANs are a complex assignment, that often suffers from multiple problems, most common examples are mode-collapse and the networks never being able to converge. This has led to the design and experimentation of a multitude of GAN versions, one of these versions are the Wasserstein GAN. The version is supposed to improve the system by smoothing the gradient and enabling better learning from both successful and unsuccessful generations. [61]

The version introduces a new loss function called the Wasserstein loss. It is based on the Wasserstein metric and first proposed in the 2017 paper "Wasserstein GAN" [62]. Instead of limiting, by using a threshold of 0.5 to give a class prediction between real and fake (1 and 0), the function outputs a scaled number as a score from the discriminator [57]. The number is not bounded between  $[0, 1]$ , instead it calculates values that represent real instances with larger values and smaller values for fake instances. This process does not discriminate, providing more of a critics opinion about the result [60]. Therefore, the discriminator is a critic in this GAN version, regardless it is only of theoretical importance, the practical application is similar between the discriminator and critic.

For the Wasserstein loss function to create the scaled numbered score " $f$ ", the Sigmoid function is removed from the critic (discriminator). The Wasserstein function is difficult to compute and handle efficiently, therefore it has been simplified with the Kantorovich-Rubinstein duality. The simplification results in the finding of the least upper bound, forcing the application of a "1-Lipschitz function" as constraint (equation 2.5) on the gradient equation for the critic. To enforce the constraint, clipping is introduced, constraining the critic-gradient so that it does not exceed  $K$  in equation 2.5 [63][64].

The Wasserstein distance loss equations are implemented below, having been simplified with the Kantorovich-Rubinstein duality [60].

### Lipschitz constraint formula

$$|f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2| \quad (2.5)$$

### Clipping:

The clipping occur after every critic-gradient update, clamping the weights to a fixed range  $[-c, c]$  [61].

### Discriminator/Critic:

$$\nabla_{\omega} \cdot \frac{1}{m} \cdot \sum_{i=1}^m [f(x^{(i)}) - f(G(z^{(i)}))] \quad (2.6)$$

### Generator:

$$\nabla_{\theta} \cdot \frac{1}{m} \cdot \sum_{i=1}^m [f(G(z^{(i)}))] \quad (2.7)$$

- $f$  = Numbered score representation of how real/fake the Critic evaluate an input.
- $c$  = Clipping interval (a common value is 0.01).

## 2.6 Wasserstein Generative Adversarial Network with Gradient Penalty (WGAN-GP)

The solution proposed in WGAN has problems with sometimes only generating poor samples, slow training and a difficulty converging, in addition to not showing obvious improvements compared with the original GAN, during real experiments [10][64]. An improvement was proposed in the fittingly titled paper "Improved Training of Wasserstein GANs" [10], that identified problems to be related to WGANs direct application of weight clipping to achieve the Lipschitz constraints. Additionally, the clipping has a very sensitive hyper-parameter ( $c$ ), prone to cause problems that result in exploding or disappearing gradients.

The paper proposes using gradient penalty to enforce the Lipschitz constraints, reasoning with that the method does not suffer the same problems as weight clipping. It is implemented by adding the formula as an extra loss term, to the loss function in the discriminator/critic. The 1-Lipschitz constraints are satisfied if the gradients ( $f$ ) have a norm of maximum 1 [60].

**Gradient penalty:**

$$\gamma \cdot \frac{1}{A} \cdot \sum_{a=0}^1 (\|f(a \cdot G(z)) + f(x - a \cdot x)\|_2 - 1)^2 \quad (2.8)$$

**Discriminator/Critic:**

$$\nabla_{\omega} \cdot \frac{1}{m} \cdot \sum_{i=1}^m \left[ f(x^{(i)}) - f(G(z^{(i)})) + \gamma \cdot \frac{1}{A} \cdot \sum_{a=0}^1 [(\|f(a \cdot G(z)) + f(x - a \cdot x)\|_2 - 1)^2] \right] \quad (2.9)$$

**Generator:**

$$\nabla_{\theta} \cdot \frac{1}{m} \cdot \sum_{i=1}^m [f(G(z^{(i)}))] \quad (2.10)$$

- $\gamma$  = Gradient penalty coefficient.
- $a$  = Variable for calculating the uniformly sampled values of  $G(z)$  and  $x$ , between 1 and 0 [60].
- $A$  = Amount of intervals of  $a$  between 0 and 1.

Since Batch Normalization crates correlations between samples in the same batch, it is not used with the gradient penalty. The reason is that it impacts the effectiveness, therefore Layer Normalization is used as mentioned in chapter 2.2.4 [60].

## 2.7 Related work

The original "Generative Adversarial Nets" [9] proposed in 2014, showed promising innovation for a new method of training generator models to generate artificial data from a collection of data. The data created had the possibility of being new variations of the data in the collection.

With further research based on the original GAN, the paper "Unsupervised representation learning with Deep Convolutional Generative Adversarial Networks" [31] introduced a GAN based on convolutional neural networks called DCGAN in 2015. This version proved to have more stable models with better prediction power. It was noted that further work is needed to improve the stability further as the experimentation showed oscillations.

In recent years, further research into improving GAN has become a popular subject, leading to the creation of proposed extensions and advancements (explanation inspired by source [65]). The extension-versions use the GAN and DCGAN as foundation for proposing different architecture or training processes, some examples are StackGAN [66] and cGAN [67]. Further building on the foundations of evaluation-version are the advanced-versions that have showed impressive results, some examples are WGAN [62] and CycleGAN [68].

Machine learning libraries like TensorFlow [69] and PyTorch [70] has realized the communities interest in experimenting with GANs. They have thereby introduced methods to help simplify the implementation of GAN in programming languages like Python [71]. A drawback is that these complex deep neural networks need powerful hardware, usually provided by a GPU. This require the enabling of GPU support for the libraries, which may be a daunting task (further elaborated in chapter 3).

A problem with the creation of GANs are that of evaluating them. There are created many proposed methods trying to solve this problem, there are 29 proposed methods evaluated at source [72]. There are some widely used methods for comparing GANs on specific datasets, but there are still limitations for evaluating generalized performance.

The paper "Improved training of Wasserstein GANs" [10] introduced the WGAN-GP in 2017. This version have performed well in tests that compare the performance between multiple GAN versions, the test can be found at source [73], where the tests on the CELEBA (faces) and CIFAR-10 (animals and vehicles) datasets, are higher regarded than tests for the datasets of MINST (symbols and numbers) and FASION-minst (greyscale fashion products) [74]. Reasoning with these datasets being more similar to real life images and especially images of real objects.

Research of automatic object detection of previously unseen images, as preformed in "Detecting previously unseen objects without human intervention using neural networks." [16]. As that research advances, it might be time for exploring the possibility of also starting with a previously unseen image of objects, then automatically creating artificial images containing different arrangement of the objects.

From my research I have not found any previous applications or testing of GANs, for the problem described in this thesis.

## 2.8 Current approach

The control system for the automatic robotic order fulfillment station is a neural network. The network can be trained with data to become able to handle objects. The data that is used for training, consists of images, labels and annotations, hereafter called a dataset.

Creating a dataset sufficient to train the neural network for object handling is currently a process that require human labor. For the thesis "Detecting previously unseen objects without human intervention using neural networks." [16] written on a requested assignment from Pickr.AI. The author created a dataset for training and experimenting on a similar neural network as the one used for the automatic robotic order fulfillment station.

For the created dataset, the author has noted, that each image on average took 5 minutes to annotate and label, using a program called COCO annotator [75]. Completing the whole dataset with 2175 images of 15 different objects took 22.5 workdays (if one workday is eight hours) [16]. Additionally, the creator writes that it took 11 hours taking the 2175 images of objects with different orientations.

The problem proposed in this thesis is an attempt at reducing the amount of images needed for the dataset, by creating artificial images of objects with different orientations. Exploring suitable methods for this task and making the theory easier to understand and implement, will hopefully establish a better foundation for further exploration. Furthermore, if combined with methods for automatic object detection of unseen objects (explored by the aforementioned thesis [16]), it might become a powerful tool for reducing manual registration efforts associated with the introducing of new objects for picking.

# 3. Methods

This chapter covers the proposed solutions and the preparations required for testing the proposed solution in chapter 4, including equipment and software setup, datasets, code implementation and evaluations.

The first two sub-chapters (3.1.1 and 3.2) will cover the setup needed to run GPU enabled software with the newest TensorFlow v2.4 library. The setup may be difficult to figure out, sources might be difficult to find and compatibility between programs versions is not a certainty. Therefore, this thesis will provide a complete setup that will work for most computers with Windows 10 and a Nvidia GPU.

## 3.1 Equipment

Training neural networks is a hefty computational task, it therefor requires hardware with high computational power to complete the training in reasonable time. The amount of computational power and temporary memory will determine the speed of training and enable training of datasets with a higher resolution.

### 3.1.1 Local computer:

To run the code for longer duration of time, a local home-computer will be used, that originally was built with the intent to run games and video editing. The specifications of the machine are listed below:

Type of component	Brand and model
CPU:	Intel core I7-8700k 3.70 GHz boxed CPU
CPU cooler:	Cooler Master masterliquid ML240L RGB
GPU:	Asus Geforce GTX 1080TI Strix 11GB GPU
Motherboard:	Asus ROG Strix Z370-E Gaming MB
RAM:	4 units of Corsair Vengeance RGB Pro DDR4 3200MHz 8GB
SSD:	Samsung 970 EVO 500GB M.2 SSD
PSU:	Cooler Master V750 750W 80 Plus Gold PSU
OS:	Windows 10 Pro 64-bit

Table 3.1: Hardware specifications of local computer used.

This computer is suitable for training neural network as it has a strong GPU, listed with a Nvidia compute capability of 6.1 at source [76]. It also has a good water-cooled CPU paired with four units of RAM with a total size of 32GB, in a well-ventilated cabinet, enabling quad channel RAM bandwidth and ensuring low temperatures during long periods of heavy load.

Hardware requirements for utilizing Tensorflow GPU can be found at [77], listing the following:

- NVIDIA® GPU card with CUDA® architectures 3.5, 5.0, 6.0, 7.0, 7.5, 8.0 and higher than 8.0. See the list of CUDA®-enabled GPU cards
- For GPUs with unsupported CUDA® architectures, or to avoid JIT compilation from PTX, or to use different versions of the NVIDIA® libraries, see the Linux build from source guide.
- Packages do not contain PTX code except for the latest supported CUDA® architecture; therefore, TensorFlow fails to load on older GPUs when `CUDA_FORCE_PTX_JIT=1` is set. (See Application Compatibility for details.)

### 3.1.2 Google Colaboratory (Colab)

Google Colab is a very useful tool for testing different examples from different creators. It is a virtual environment that runs a Python code on servers hosted by Google, directly through the browser. It has the mostly used python libraries already installed on the servers and free access to enable GPU support, which gives the user a powerful tool that can be used without configurations. As it does not require configurations and are free, it is easy to share code, leading to many websites having enabled one-click access to run their code yourself with Colab.

Many examples online require specific libraries and specific versions of these libraries to run correctly. In these cases, the libraries can be chosen or installed directly into Colab without complications (this can be very difficult on a local machine, especially while maintaining a GPU enabled). A Colab environment created by the user and the memory used, will be deleted if idle for a period of time, therefore it is normal to have code that specify the environment at the start of the code-file, as well as mounting a Google Drive to read and write long-term memory files.

Downsides with Colab is the maximum limited time of 12 hours that a virtual environment can be connected to the servers (for normal CPU computations) [78]. Further there are dynamic resource limits, based on fluctuations in demand. GPU and TBU (Tensor Processing Unit) are more limited resources than the CPU, therefore these resources are prioritized for interactive activity rather than long-running computations. This makes it less ideal for training neural networks, in experience the GPU resources are available for 3-6 hours weekdays and up to 12 hours on weekends.

## 3.2 Software

### 3.2.1 Setting up the Integrated Development Environment (IDE)

In order to run GAN training, it is ideal to use a well-developed Python ecosystem like Tensorflow [69] that is an end-to-end open source platform for machine learning. While researching the topic of GANs, it was found as the most frequently used platform for GAN generation. It's a flexible ecosystem of tools and libraries, that allows easier creation of powerful machine learning application [69].

It is usually ideal to use the newest stable version of software, the choice therefor is Tensorflow 2.4. The software also has support for utilizing one or more GPU's with the use of Nvidia CUDA software, this will increase the computational power of the hardware. Both Tensorflow and CUDA are dynamic software, meaning that updates with newer releases can be unstable and can break previously functional software combinations [79].

After trial and error, sources ([79] and [80]) where found that recommend using CUDA 11.0, explicitly stating not to use the newer releases CUDA 11.1 and 11.2. Then combining it with the cuDNN 8.0.4 as it works with Tensorflow 2.4. Using Miniconda 3 instead of the full Anaconda package was chosen as the full package could create conflicts with the installation of the previously stated software, a problem experienced as it happened in one of the first attempts of installment. The choice of Python version is irrelevant between the three Tensorflow supported versions 3.6, 3.7 and 3.8 [81]. The newest version of Python is chosen along with the usage of the Pycharm Professional IDE, since the program is free for students and the program I'm most familiar with.

#### **Python 3:**

Specific installment "3.8.8" from <https://www.python.org/downloads/release/python-388/> the Windows 64-bit installer.

#### **Miniconda 3:**

Specific installment "Python 3.8 - Miniconda3 Windows 64bit" from <https://docs.conda.io/en/latest/miniconda.html>. Normal installation with the following advanced options added: "Add Anaconda to my PATH environment variable." and "Register Anaconda as my default Python 3.8".

#### **Pycharm professional:**

Specific installment "Pycharm 2020.3.5" from <https://www.jetbrains.com/pycharm/> the Windows 64-bit launcher. Software license activated with a student account.



### 3.2.2 Tensorflow GPU support:

Software requirements for utilizing Tensorflow GPU can be found at [77], listing the following:

- NVIDIA® GPU drivers —CUDA® 11.0 requires 450.x or higher.
- CUDA® Toolkit —TensorFlow supports CUDA® 11 (TensorFlow >= 2.4.0)
- CUPTI ships with the CUDA® Toolkit.
- cuDNN SDK 8.0.4 cuDNN versions.
- (Optional) TensorRT 6.0 to improve latency and throughput for inference on some models.

#### Visual Studio Community 2019:

This program is a prerequisite for installing the CUDA toolkit [82], installed from <https://visualstudio.microsoft.com/>. Only the program is installed, no need for the program's workload packages.

#### CUDA Toolkit 11.0 Update 1

Specific installment "cuda\_11.0.3\_451.82\_win10.exe" from [https://developer.nvidia.com/cuda-11.0-update1-download-archive?target\\_os=Windows&target\\_arch=x86\\_64&target\\_version=10&target\\_type=exelocal](https://developer.nvidia.com/cuda-11.0-update1-download-archive?target_os=Windows&target_arch=x86_64&target_version=10&target_type=exelocal).

#### cuDNN

Specific installment "cuDNN v8.0.4 (September 28th, 2020), for CUDA 11.0" from <https://developer.nvidia.com/rdp/cudnn-archive> the "cuDNN Library for Windows (x86)" download. The zip-file is extracted to the file location: C:\\tools\\cuda.

### 3.2.3 Virtual environment for the local computer

Taking inspiration and following the recommendations from sources [79] and [80], to set up the virtual environment. Firstly the following paths (in figure 3.1) have to be set into environment variables (system variables) for CUDA 11 to function in Windows 10 (in addition to the paths created during the installation).

```
C:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v11.0  
C:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v11.0\\extras\\CUPTI\\lib64  
C:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v11.0\\include  
C:\\tools\\cuda\\bin
```

Figure 3.1: Showing the path's needed for this environment.

Open Anaconda prompt (Miniconda 3) and run the following commands, to create environment:

```
"conda create -c conda-forge python=3.8.5 -n ProjectName"
```

(choosing the 3.8.5 version of the installed python 3.8 because of recommendations).

Activate environment:

```
"conda activate ProjectName"
```

Install Jupyter notebook package:

```
"conda install -y jupyter"
```

Add Jupyter support to your new environment:

```
"conda install nb_conda"
```

It is also be a good idea to install the following popular packages at this point (required for the code in this thesis):

tensorflow, glob2, imageio, matplotlib, numpy, pillow, scikit-image, oyaml, OpenCV, func-tools and tqdm.

**Side note:** From Tensorflow version 2.0+, the libraries tensorflow and tensorflow-gpu are no longer separate. It is only required to install tensorflow as it includes GPU support if appropriate graphic card and CUDA software are installed [77] [83].

Create a new kernel:

```
"python -m ipykernel install --user --name ProjectName --display-name "Python 3.8 (ProjectName)" "
```

GPU support can be tested by running the following code in Anaconda prompt:

```
"python"
"import tensorflow as tf"
"tf.__version__"
"gpu = len(tf.config.list_physical_devices('GPU'))>0"
"print("GPU is", "available" if gpu else "NOT AVAILABLE")"
```

It can also be tested in Pycharm as shown in figure 3.2.



```
1  ▶  #%%
2
3  import tensorflow as tf
4  tf.__version__
5
6  ▶  #%%
7
8  print(f"Tensor Flow Version: {tf.__version__}")
9
10 ▶  #%%
11
12 gpu = len(tf.config.list_physical_devices('GPU'))>0
13 print("GPU is", "available" if gpu else "NOT AVAILABLE")
14
15
```

```
1  import tensorflow as tf
   tf.__version__
1  '2.4.1'
2  print(f"Tensor Flow Version: {tf.__version__}")
   Tensor Flow Version: 2.4.1
3  gpu = len(tf.config.list_physical_devices('GPU'))>0
   print("GPU is", "available" if gpu else "NOT AVAILABLE")
   GPU is available
```

Figure 3.2: Showing the GPU support test output in Pycharm.

The environment and kernel can now be launched in the browser by running command **"jupyter notebook"** or opening Pycharm with the environment and kernel created.

## 3.3 Proposed solution

The solution for this thesis will explore if Generative Adversarial Networks, may be suitable to generate more images for a dataset containing a limited amount of images with previously unseen objects. It will firstly determine if the original DCGAN is able to perform this task, seeing as it is a simpler computational method, therefore a less time-consuming method, that is supposed to provide generated images of good quality. There are some expected problems with this method as mentioned in chapters 2.4 to 2.6.

Secondly the more advanced version WGAN-GP will be explored, this version is a proposed solution to the problems of the WGAN, a subsequent improvement of the DCGAN. This version is shown to be a well performing GAN on datasets similar to the one that will be used in this thesis, this is previously explained in paragraph six of chapter 2.7.

If proven successful in generating new images from a large dataset, it will be tested on a limited dataset. The limited dataset is chosen to be of the size of ten images, on the presumption that it might be possible for the robotic automation station to be informed of the content of the next ten boxes on the conveyor belt.

Two methods will be attempted to try and improve the generated images, pre-training the model on a large set of images before training on unseen images and retraining the model on an expanded dataset of generated images. Additionally, it will be shown how the GAN perform on a dataset of twenty images.

Object detection software is used for a network to handle objects at robotic order fulfilment stations. An object detector will be trained on real images and tested on generated artificial images, to confirm if the images are sufficient for object detection and ensure that the object detector has no unforeseen problems with these artificially generated images.

As a part of the solution is to determine if Generative Adversarial Networks may provide a suitable solution. An effort is made to make this relatively new research field easier to navigate, and to visualize new areas where this technology might be utilized.

### 3.3.1 Object detector

The solution is supposed to be used in combination with objection detection software. Therefore, it will be of interest to confirm that the generated images are of sufficient quality to be detected.

#### **Experiment: Train and test YOLOv5x**

A popular and well performing object detection software called YOLOv5x will be trained and tested on the dataset, to show how well it can perform with data given in the similar manner as the current approach of training.

#### **Experiment: Detect generated objects with YOLOv5x**

After performing all the other experiments listed in this sub-chapter, the best generated images (from experimentation) will be tested with the object detection software YOLOv5x.

### 3.3.2 GAN (DCGAN) vs. WGAN-GP

Experiments will be performed to confirm that a more advanced version of GAN is needed. This is because the GAN is supposed to be easier to run and provide good quality images, thereby reducing time required for experiments. This test will be performed on the largest dataset group (all the images provided of the same object) available in the dataset used for this thesis. The test will be able to show model stability, in addition to how time and epochs will affect the quality of images generated. This test should show if Generative Adversarial Networks are able to generate new artificial images, that does not already exist in the dataset it is trained on.

#### **Experiment: GAN generated objects**

The purpose of this experiment is to find out how the GAN (DCGAN) perform on a large dataset with images of boxes containing objects in different positions and orientations.

#### **Experiment: WGAN-GP generated objects**

The purpose of this experiment is to find out how the WGAN-GP perform on a large dataset with images of boxes containing objects in different positions and orientations.

### 3.3.3 Generate images from a limited dataset

The main experiment to be performed in this thesis, the goal is to generate new images of sufficient quality, so that the objects in the image may be recognized by object detection software (YOLOv5x). Additionally, the experiment will show if Generative Adversarial Networks may be a suitable method for expanding a limited dataset with new additional images.

#### **Experiment: WGAN-GP with 10 images**

Ten images of objects with different orientation will be used to train a WGAN-GP and see how well it perform.

#### **Experiment: Testing pre-training**

This experiment is to examine if a WGAN-GP may be pre-trained.

#### **Experiment: Pre-trained with 10 images**

This experiment is to examine how pre-training a WGAN-GP, before training it for the same amount as the previous experiment, will affect the generated images.

#### **Experiment: Training with generated images in the dataset**

This experiment is to examine how expanding the dataset with already generated images from the previous experiment, will affect the generated images.

#### **Experiment: Expanding dataset with 10 images**

The purpose of this experiment is to find out how the WGAN-GP perform with a dataset of twenty real images.

### 3.4 Dataset

The supplied dataset for this thesis was collected for a previous master thesis ([16]) written on behalf of Pickr.AI and given the name "RU Beiersdorf dataset". It consists of 2151 RGB images of 14 different objects, split into 15 groups of  $\approx 150(\pm 1)$  images, with one of the 15 groups having 50 images with a mix of objects from two groups (image 4 in figure 3.3). The file format is .png with a resolution of  $1280 \times 720$  and a bit depth of 24. The objects have been manually outlined in the images by the creator, an Icelandic master student named Sverrir Bjarnason. This makes it possible to access the labels and bounding boxes at the precise locations of the objects. The RU Beiersdorf dataset are redistribution prohibited without written consent [84].

After some initial testing on the systems hardware (local computer 3.1.1), the system has shown to be capable of handling training with a dataset of resolution up to  $256 \times 256 \times 3$  on a batch size of 64. Higher values may sometimes result in crashes because of memory problems and does result in an overall higher temperate threshold. Therefore, a resolution of  $256 \times 256 \times 3$  is chosen as it gives the most information in the images along with the possibility of a variable batch size from 4 to 64.



Figure 3.3: Example images from all the different groups, the groups are referred to with numbers starting at the top left corner (top row is 1, 2 and 3).

### 3.4.1 Dataset for object detector

The labels of the dataset are in the file format COCO (.json), these were not readable for the YOLOv5 code and not readable for an image and label converter application named Roboflow [85]. Therefore, the code from [86] was downloaded and edited to convert the images and labels into "YOLO (darknet)" format. After the conversion the dataset was readable when uploaded to Roboflow, the application provides services for conversion into almost every format desirable, in addition to resizing, augmenting, splitting etc. It was acquired in the formats of COCO, CreateML, TFRecord, VOC, YOLO (keras) and YOLOv5 (PyTorch), where the YOLOv5 version is the one to be used.

#### Distribution and augmentation:

The YOLOv5 formatted dataset will be split into three sets, training, validation and test, with a standard distribution of 70%, 20% and 10%. There are many more instances than the minimum [87] of 10 per class, there are about 600-800 instances per group and some groups belong to the same class. Therefore, no augmentation is applied to this dataset.

#### Image format:

The file format from Robotflow is .jpg, a different file format than the .png of the original dataset. The differentiation between the file formats are not relevant, as neural network models transform the information in the images into arrays or tensors for training, which is the case for the models used in this thesis (GAN and YOLOv5).

#### Examples of annotated images:

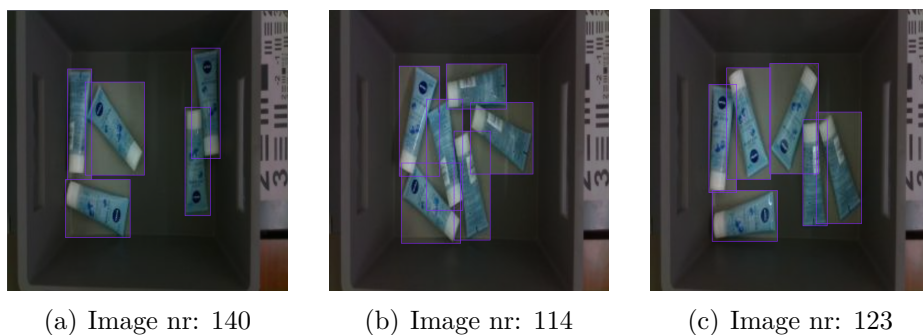


Figure 3.4: Showing annotated images of the same product images that are going to be used for GAN generation.

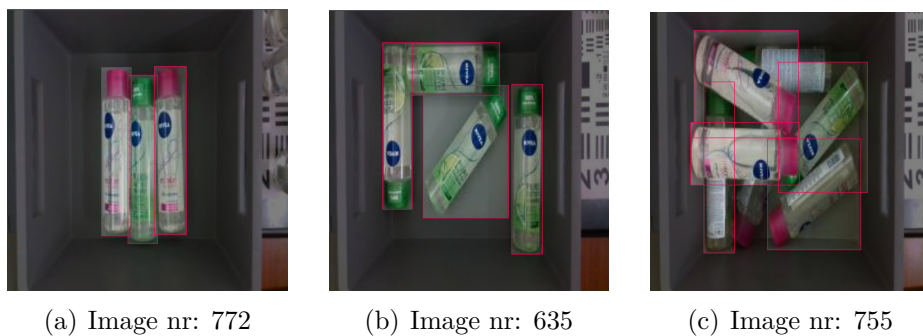


Figure 3.5: Examples of other images and objects in the dataset.

### Health check of the dataset:

A health check using Robotflow, found one fault and one limitation of the dataset. As shown in figure 3.6 the class 0 representing "Bin" as environment (information about classes from the original annotations[84]), are underrepresented with only seven instances. Deeper investigation revealed that there only was one instance of class 0 in the dataset, the other six instances was caused by multiple misclassifications in one single image (nr: 2121). The objects in this image was reclassified as class 3. The dataset now only contains one last instance of class 0 (image nr: 912), this is not relevant on the basis of the object detector being used to recognize generated objects, where the environmental class is stationary and of little importance.

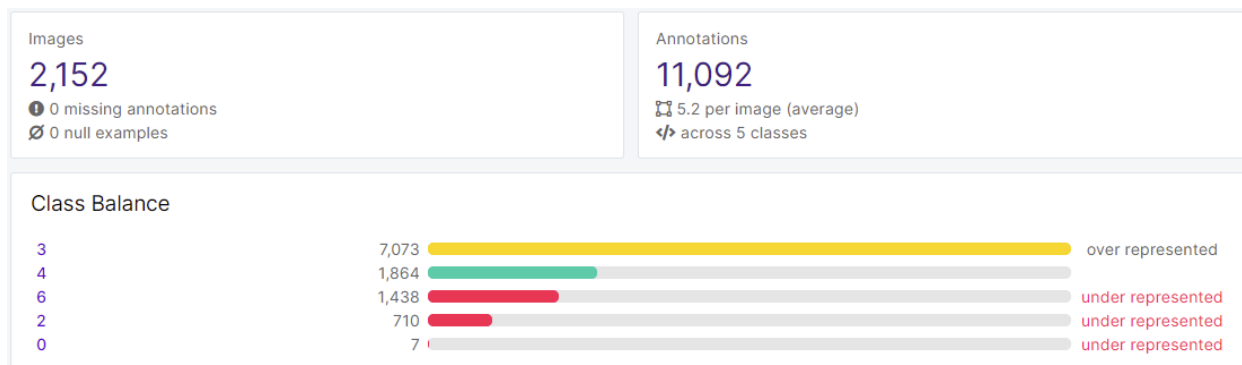


Figure 3.6: Health check performed in the application Roboflow.

The remaining classes 2 (Tupur), 3 (Bottle), 4 (Can) and 6 (Spray), are in the category of products. The class names will be changed to letters, on account of YOLOv5 returning predictions as two parameters: "class prediction". A letter for class and a numbered prediction will show a clearer result. Class 0 will be "E", followed by the other classes as "T"(2), "B"(3), "C"(4) and "S"(6). Note that class 1 and 6, listed as "product" and "environment", was not found to be classified with bounding boxes in the dataset.

### 3.4.2 Dataset for GAN generation

It will be used five versions of the dataset. The first version will be the whole complete dataset with 2151 images, the next two versions are all the images from two groups of the different objects, the last two will be smaller portion of the two previous versions. The two object groups shown in figure 3.7 are chosen on the premise of having distinct features, not being transparent or having a metallic surface, not having mixed grouping and including a wide array of object orientations in their groups.



(a) Image of object group 1 (image nr 110). (b) Image of object group 4 (image nr 585).

Figure 3.7: Examples from the two smaller datasets.

All the versions of this dataset are considered as a "small datasets", compared with the most common datasets used for GAN training. This is significant to point out as the training of a GAN are reliant on much data, equal to most machine learning processes, it benefits from more data. The dataset I have found to be applied the most is CIFER-10 dataset, containing 10 groups/classes comprised of 6000 images in each [88]. The dataset for this thesis is somewhat more comparable to the CIFAR-100, with 100 groups comprised of 600 images each. Because where the CIFAR-100 has 600 images of one single object, this dataset has  $\approx 150$  images with an average of  $\approx 4.63$  objects in each (calculated from the values in the table below), totaling 725 and 670 visualizations of the objects.

Number of objects in the image:	Dataset of object 1	Dataset of object 4
1	13	8
2	10	10
3	14	18
4	11	30
5	12	36
6	91	48
Total number of images in dataset:	151	150
Total number of objects in dataset:	725	670



It is possible to create GAN and GAN versions on the CIFAR-100 dataset (multiple versions created in the paper [89]). For this dataset there will be a difference in how the objects are delivered to the GAN. The resolution will be higher 256\*256 compared to 32\*32 (CIFAR-100), with multiple objects in each image, compared to one in each. Delivering the information about the objects with the method described is found to be unconventional and mostly untested by the GAN creators community.

### **Distribution and augmentation:**

The dataset for GAN does not rely on distribution, there are no need for validation or test images. The generator model is not connected directly to the dataset and will not be able to know the information from the real images. The focus will be on generations of objects in the gray bin/box, the discriminator should understand the difference between stationary (box) and movable objects (products). Therefore, there will not be used augmentations, a known method for improving small datasets, without supplementing them. The augmentations would have affected the whole image, for example distorting, flipping, rotating and moving the box. The dataset contains different arrangement of objects, not environments (box), the hope is that the GAN will create more arrangement of the objects, without changing the environment.

### **Image format:**

The format of images .png, as mentioned earlier, this is of little importance for a neural network. It is a "community standard" of using square pixelated images, the same will be done for these experiments. The reasoning is that it is easier to keep track of square data sizes and uniform layers, a correction will yield little more than a satisfactorily feeling of correct proportions. It is worth noting, implement the correct image ratio may be performed by editing the stride and fractional stride in the convolutional layers. This will enable the use and generation of rectangular images, as the strides can have different values for horizontal and vertical axis, making it possible to create the desired pixel sized image.

## 3.5 Object detector YOLOv5

YouOnlyLookOnce version 5 (YOLOv5) is an object detection application in python, build on Pytorch libraries. The application provide a framework for training and using object detectors. Additionally, it provide pre-trained models of different complexity. These models increase the speed of training for new objects and will be able to handle some objects without training.

### 3.5.1 Code for YOLOv5

The code for using YOLOv5 is a simple one, consisting of downloading the developers resource library from GitHub at [90] and then installing the "requirements.txt" file. The library contains a pre-trained models for object detecting, after importing Image and clear\_output (both from IPython.display) along with importing torch, it is possible to specifically train the model or to use it for detecting, with one line of code (tutorial are found at [91]).

Since YOLOv5 utilize Pytorch instead of Tensorflow and the training are a less time consuming operation than the training of GANs, Google Colab (3.1.2) will be used for this task. This will ensure that local computer GPU-environment is not affected by other software and enable the possibility of simultaneous training and usage.

The pre-trained model YOLOv5x (specifically YOLOv5x5, because of pixel size [8]), will be implemented and tested for 100 epochs, with GPU enabled in Google Colab. The batch size will be set at 16, as this is the recommended value for 16 GB GPUs [8]. The type of GPU encountered when using Colab may vary for every reconnection, but in experience they are usually  $\approx 16$  GB or above. The reasoning for choosing YOLOv5x is shown in figure 3.8, where it is the best performing model. If the complexity of the model give rise to problems like time constraints or computational problems, the alternation into a less computational model can be made.

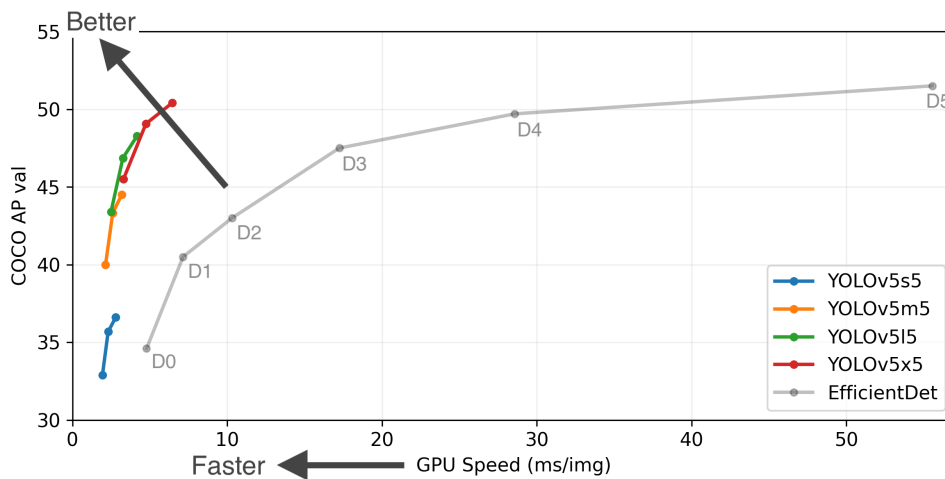


Figure 3.8: Shows the performance of the YOLOv5 edition 5 models. "COCO AP val" is a value representing the averaged over all categories [7], calculating multiple intersection union between a detected area and an annotated area (validation dataset), this explanation is only valid for the COCO context (elaborated at source [7]). "GPU speed" self-evident as the milliseconds used per image, during training. Image from source [8].

## 3.6 Code for GAN generation

After a lot of trial and error, source material for the code was found at GitHub [92] from the creator Zhenliang He. He is a PhD student at the Institute of Computing Technology, Chinese Academy of Sciences [93], who has released his work under an MIT license [94]. The code for this thesis is built from python scrips provided in the previously mentioned GitHub repository, into a single Jupiter notebook file. The code will be provided in the appendix, it provides a simple overview to select parameters and GAN version.

The optimizer used in the experiments will be Adaptive Moment Estimation (ADAM), although it is originally only implemented in WGAN-GP. The choice of this optimizer is explained in chapter 2.3.3. This optimizer will help with stability over long training periods and introduce more generalization between the GAN versions. It is worth noting that it is easy to change the optimizer if required and to emphasize that this code incorporate many versions of GANs (GAN, HingeGAN, LSGAN, WGAN-GP, DRAGAN).

### ADAM:

The optimizer will be implemented with the following values (where most are Tensorflow default values listed at source [95], the source also provides a deeper description of each parameter):

- *Learning rate* ( $lr$ ) = 0.0002 (not default)
- $\beta_1 = 0.5$  (not default)
- $\beta_2 = 0.999$
- $\epsilon = 1 \cdot e^{-7}$
- *amsgrad* = *False*
- *name* = *Adam*
- *\*\*kwargs*

In the pseudocode (shown in figure 3.10) of the original paper for WGAN-GP the  $\beta_1 = 0$  representing zero gradient averaging, but in the appendix (page 14 E) the value  $\beta_1 = 0.5$  is also used for testing. The paper [55] state that the most common values for  $\beta_1$  with ADAM in GAN is from 0 to 0.5. Additionally, noting that no rationale is given for this choice. The values  $\beta_1 = 0.5$  and  $lr = 0.0002$  is recommended with little reason given at sources [30] but are noted to help with stabilizing in the paper [31]. Furthermore, in the paper [55] a test of relations between  $\beta_1$  and batch size is preformed, the test shows signs of this correlation between the parameters (tested at batch size 64 and 128). The test is interpreted to have bad results for small  $\beta_1$  values (0.1 bad for both batch sizes) and unpredictable result for large values (0.8, good for batch size 64 and bad for batch size 128). The authors conclude that: "Our initial investigations have shown that larger batch sizes can allow for lower  $\beta_1$  values.". For the experiments in this thesis a batch size of 64 or lower will be used for all test, based on this research the value for  $\beta_1$  and  $lr$  are set to 0.5 and 0.0002 as a compromise that incorporate the smaller batch sizes.

### Batch size, epochs and iterations:

The batch size will be high at 64, for the initial experiments, for the two datasets of  $\approx 150$  images this will give two iterations per epoch. A small amount of iteration per epoch will give a faster training, enabling the possibility of training for more epochs. The consequence of a high batch size is a lower accuracy of the estimate of the error gradient, there is also a trade-off between speed and stability of training [96]. A large batch size will give a faster result, knowing that the accuracy and stability can be improved, if the experiments show promising results. As mentioned earlier (in chapter 3.4), the highest batch size for the setup in this thesis, is tested to be 64 for images with a pixel value of  $256*256*3$ . Smaller values for this parameter must be used if there are less than 64 images in the dataset.

### GAN:

Pseudocode for the GAN training algorithm from the original GAN paper [9] is shown in figure 3.9. For this thesis ADAM will be used instead of momentum, for the experiments. The pseudocode describe how the training of the Discriminator and Generator will be implemented, the structured layout of these model is shown in the table 3.2 (Discriminator) and 3.3 (Generator).

---

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

---

**for** number of training iterations **do**

**for**  $k$  steps **do**

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(\mathbf{x}^{(i)}) + \log \left( 1 - D(G(\mathbf{z}^{(i)})) \right) \right].$$

**end for**

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left( 1 - D(G(\mathbf{z}^{(i)})) \right).$$

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

---

Figure 3.9: Pseudocode for the GAN training algorithm, taken from the original paper [9]. The inner "for"-loop minimizes the function of the discriminator, the outer "for"-loop maximizes the function of the generator, leading to alternation training between the models (one at the time).  $k$  is notated as  $N_D$  in the code ( $n_{\text{critic}}$  in figure 3.10), stating how many times the discriminator will run the training loop for every generator loop (equal to the number of epochs).

### WGAN:

The WGAN will not be implemented, as it is only used as a tool to understand the inner workings of Improved Training of Wasserstein GANs (WGAN-GP) in chapter 2.5. The reason is, that the GAN (DCGAN), is noted to have slightly better image quality and be of simpler complexity that leads to a faster convergence than both WGAN and WGAN-GP [60]. Furthermore, the WGAN-GP receives better or marginally similar result during tests, has better stability and a faster convergence, when compared to the WGAN.

Side note: if interested in the WGAN, source code can be found at [92] in the version 1 folder. It is not implemented, at least not correctly in version 2.

### WGAN-GP:

The WGAN-GP is implemented as shown in 3.10, with the new values for ADAM. Pseudocode for the WGAN-GP training algorithm from the Improved Training of Wasserstein GANs paper [10] is shown in figure 3.10. For this thesis ADAM will be implemented with the values mentioned earlier in this chapter. The pseudocode describe how the training of the Critic and Generator will be implemented, the structured layout of these model is shown in the table 3.2 (Critic) and 3.3 (Generator).

---

**Algorithm 1** WGAN with gradient penalty. We use default values of  $\lambda = 10$ ,  $n_{\text{critic}} = 5$ ,  $\alpha = 0.0001$ ,  $\beta_1 = 0$ ,  $\beta_2 = 0.9$ .

---

**Require:** The gradient penalty coefficient  $\lambda$ , the number of critic iterations per generator iteration  $n_{\text{critic}}$ , the batch size  $m$ , Adam hyperparameters  $\alpha, \beta_1, \beta_2$ .

**Require:** initial critic parameters  $w_0$ , initial generator parameters  $\theta_0$ .

```
1: while  $\theta$  has not converged do
2:   for  $t = 1, \dots, n_{\text{critic}}$  do
3:     for  $i = 1, \dots, m$  do
4:       Sample real data  $\mathbf{x} \sim \mathbb{P}_r$ , latent variable  $\mathbf{z} \sim p(\mathbf{z})$ , a random number  $\epsilon \sim U[0, 1]$ .
5:        $\tilde{\mathbf{x}} \leftarrow G_{\theta}(\mathbf{z})$ 
6:        $\hat{\mathbf{x}} \leftarrow \epsilon \mathbf{x} + (1 - \epsilon) \tilde{\mathbf{x}}$ 
7:        $L^{(i)} \leftarrow D_w(\tilde{\mathbf{x}}) - D_w(\mathbf{x}) + \lambda(\|\nabla_{\hat{\mathbf{x}}} D_w(\hat{\mathbf{x}})\|_2 - 1)^2$ 
8:     end for
9:      $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$ 
10:  end for
11:  Sample a batch of latent variables  $\{\mathbf{z}^{(i)}\}_{i=1}^m \sim p(\mathbf{z})$ .
12:   $\theta \leftarrow \text{Adam}(\nabla_{\theta} \frac{1}{m} \sum_{i=1}^m -D_w(G_{\theta}(\mathbf{z})), \theta, \alpha, \beta_1, \beta_2)$ 
13: end while
```

---

Figure 3.10: Pseudocode for the WGAN-GP training algorithm, taken from the paper [10].  $n_{\text{critic}}$  is notated as  $N_D$  in the code, stating how many times the critic will run the training loop for every generator loop (equal to the number of epochs). The random number  $\epsilon$  is implemented as  $\sim N(0, 1)$  not  $\sim U(0, 1)$ . The inner "for"-loop calculate the gradient penalty, the outer "for"-loop maximizes the function of the critic and the "while"-loop maximizes the function of the generator. The training alternates between the models (one at the time). Formula for  $L^i$  is the same the simplified formula 2.9 in chapter 2.6

Discriminator/Critic layers:	Parameters for the layer
Input:	256x256x3 image
Convolutional:	dim = 256*256*3, filter size = 4*4 and stride = 2
LeakyReLU:	dim = 128*128*64 and $\alpha=0.2$
Convolutional:	dim = 128*128*64, filter size = 4*4 and stride = 2
Batch/Layer Normalization:	dim = 64*64*128
LeakyReLU:	dim = 64*64*128 and $\alpha=0.2$
Convolutional:	dim = 64*64*128, filter size = 4*4 and stride = 2
Batch/Layer Normalization:	dim = 32*32*256
LeakyReLU:	dim = 32*32*256 and $\alpha=0.2$
Convolutional:	dim = 32*32*256, filter size = 4*4 and stride = 2
Batch/Layer Normalization:	dim = 16*16*512
LeakyReLU:	dim = 16*16*512 and $\alpha=0.2$
Convolutional:	dim = 16*16*512, filter size = 4*4 and stride = 2
Batch/Layer Normalization:	dim = 8*8*512
LeakyReLU:	dim = 8*8*512 and $\alpha=0.2$
Convolutional:	dim = 8*8*512, filter size = 4*4 and stride = 2
Batch/Layer Normalization:	dim = 4*4*512
LeakyReLU:	dim = 4*4*512 and $\alpha=0.2$
Convolutional:	dim = 4*4*512, filter size = 4*4 and stride = 1
Output:	dim = 1*1*1, decision: real or fake/scaled numbered score

Table 3.2: Structured layout of Discriminator/Critic (the sign "/" is used to differentiate between the Discriminator of the GAN and the Critic of the WGAN-GP).

Generator layers:	Parameters for the layer
Input:	128 datapoints of a normal distribution between 1 and 0.
Convolutional transpose:	dim = 1*1*128, filter size = 4*4 and fractional stride = 2
Batch Normalization:	dim = 4*4*512
ReLU:	dim = 4*4*512 and $\alpha=0.2$
Convolutional transpose:	dim = 4*4*512, filter size = 4*4 and fractional stride = 2
Batch Normalization:	dim = 8*8*512
ReLU:	dim = 8*8*512 and $\alpha=0.2$
Convolutional transpose:	dim = 8*8*512, filter size = 4*4 and fractional stride = 2
Batch Normalization:	dim = 16*16*512
ReLU:	dim = 16*16*512 and $\alpha=0.2$
Convolutional transpose:	dim = 16*16*512, filter size = 4*4 and fractional stride = 2
Batch Normalization:	dim = 32*32*256
ReLU:	dim = 32*32*256 and $\alpha=0.2$
Convolutional transpose:	dim = 32*32*256, filter size = 4*4 and fractional stride = 2
Batch Normalization:	dim = 64*64*128
ReLU:	dim = 64*64*128 and $\alpha=0.2$
Convolutional transpose:	dim = 64*64*128, filter size = 4*4 and fractional stride = 1
Tanh:	dim = 256*256*3
Output:	Generated image of dimension 256*256*3

Table 3.3: Structured layout of Generators.

## 3.7 Methods of evaluating GANs

The evaluation of a resulting GAN is a more complex task than for many other deep neural networks. This is because where other deep neural networks usually rely on the convergence of a loss function directly connected to the model. This loss function is monitored during training and when the loss has not improved for a pre-defined number of iterations, the model has reached a convergence. As previously mentioned, (chapter 2.3.3) the discriminator is trained to discriminate between real and fake (generated), this is then used to train a generator. The GAN has completed training when the discriminator has a 50 % confidence in determining if an image is real or fake. Generator loss is created by the discriminator and not directly connected to the generator model, therefore there are no objective loss function to follow and use to assess the process [56].

A professor named Jason Brownlee has released a lot of literature on research into GAN, including the an evaluation on five scientific studies ([97], [72], [98], [99] & [100]), the evaluation can be found at source [56]. The choice of evaluation techniques will be based on his work, recommendations and research into these studies, that cover many different methods including five qualitative- and twenty-four quantitative techniques.

From this evaluation ([56]) it is found that it is best to start evaluating with "manual image inspection". Then when that evaluation show that the models improve, the move can be made to quantitative methods that can summarize the quality and diversity of the generated images. As there are no single best and agreed upon measure, the choice is made to assess the two that come the closest, "Inception Score" and "Frechet Inception Distance". The evaluation also recommends combining the quantitative methods with two quantitative methods "Nearest Neighbors" and "Rating and Preference Judgement" (human-based) to provide a robust assessment.

### 3.7.1 Manual inspection

This method is human manual inspection of the generated results. Controlling the quality of the image. Ensuring that the image is a new creation and not that the generator has learned to create the original dataset, which the model has never seen. The generator will be sampled with 100 generated image samples at regular intervals and at the end of training. Comparing one of sampled result (100 images in one) to the images in the dataset, will consist of comparing 100 individual images to the amount of images used from that dataset group (type of object). An example may be comparing a sample image of a GAN trained on one whole dataset group, will result in  $\approx 15000 (\pm 100)$  comparisons, a time-consuming process.

#### **Inspection with object detector (YOLOv5):**

Using an object detector will be a form of manual inspection of selected images, validating that the images are sufficient to be used by an artificial neural network and have no unforeseen issues.

### 3.7.2 Quantitative methods

#### **Inception Score (IS):**

An early and somewhat widely adapted evaluation method for GAN generated images. The method involves using a pre-trained model classifier to give a score on the image quality and diversity, the score has been found to be correlating well with the subjective evaluation [101]. There are many shortcomings with the IS method. It only evaluates generated images, therefore it does not compare generated and real images, not knowing if an image is truly a new creation. For multiple object generation, it will only evaluate the classes that are recognized, ignoring the partial created objects as background. The method is really only useful for conditional GANs (cGAN [102]) with clear classes and with familiar datasets for the model [103]. This is a short summary of shortcomings, information is elaborated at source [104] and the end of the video [103].

#### **Frechet Inception Distance (FID):**

This evaluation method was specifically designed to evaluate the quality of GAN generated images and has become the most widely used method for this process in recent time. It utilizes a pre-trained model and are easy to use for datasets related to the pre-trained data [105]. This is a limitation for application on a new unseen dataset, as the method is bias to the number of samples in a dataset (larger set, better values). Additionally, the method is slow to run and based on limited statistics (only mean and co-variance) when comparing the summarized score (calculated distance between the feature vectors) of real and fake images [106]. Furthermore, it is still needed to qualitatively look over the samples.

### 3.7.3 Qualitative methods

#### **Nearest Neighbors:**

This method involves comparing images with distance measurement between the pixel data [56]. It can be used for evaluating the realism of an image and to detect overfitting. The typical distance measurement used is the Euclidean distance, that is very sensitive to minor perceptual perturbations according to source [107].

#### **Rating and Preference Judgment:**

The rating and preferences of this method is based on human judgement, where several subjects are gathered to evaluate multiple results by giving them rating based on the individual preferences.

### 3.7.4 Evaluations for this thesis

For the solution proposed in this thesis, on this type of dataset, the only applicable evaluation method will be manual inspection. The quantitative methods with pre-trained models, will require a larger dataset only to train that model. What is gained from the methods are of marginal value, for this type of dataset they are both more suitable if used with a cGAN. The methods are most common to use when comparing GAN-versions trained on the same dataset. The qualitative methods are also not relevant, because of perceptual perturbations will be prevalent in the generated images and between the real ones. Additionally, the desired quality of generated images is at the level of being sufficient additions to the original dataset, not perfect additions. For this quality level, one person evaluating them, should be sufficient, especially when an object detector can be used to confirm the quality level.



## 4. Results

This chapter will present the experiments conducted and the corresponding results are presented. Three experiments are conducted, object detection (YOLOv5x, two parts), GAN vs. WGAN-GP generated images and WGAN-GP with a limited amount of images.

The presentation of results will be provided as visual results (images). The reason is that normal evaluation methods for neural networks like gradient loss, cannot objectively assess the training process and model quality for the generator model (because the loss is not directly connected to the model, as mentioned in last paragraph of chapter 2.3.3). Furthermore, the more common evaluations for GAN are deemed to provide graphs and values of minimal true value for the experiments performed in this thesis (as mentioned in chapter 3.7.4).

### 4.1 Experiment: Train and test YOLOv5x

The solution is supposed to be used in combination with objection detection software. Therefore, it will be of interest to confirm that the generated images are of sufficient quality to be detected. For this purpose, a YOLOv5x object detector will be trained in this experiment.

#### **Experiment setup:**

The most complex pre-trained version of YOLOv5, named YOLOv5x, will be used for this experiment. The model will be trained on the dataset (training and validation sets) from chapter 3.4.1 for 100 epochs, with a batch size of 16. The experiment was delegated the GPU "Nvidia Tesla K80" by Google Colab when connecting. A very powerful GPU accelerator that is designed for the most demanding computational tasks and has 24 GB (2\*12GB) memory, according to the datasheet [108].

#### **Experiment results:**

The training duration lasted for 3.17 hours. Afterwards, it was tested on 33 images that where part of the test set (previously unseen by the model). These 33 images consist of 17 images similar to (a) in figure 4.1 and 16 images similar to (b) in figure 4.1.

- (a) In the first dataset (example image illustrated in figure 4.1 (a)), all 88 objects was detected over the 17 images used, with an average certainty of 97.66% (three objects was detected but the certainty was not readable, therefore they were not included).
- (b) In the second dataset (example image illustrated in figure 4.1 (b)), 73 of 79 objects was detected over the 16 images used, with an average certainty of 95.64% (three objects was detected but the certainty was not readable, therefore they were not included).

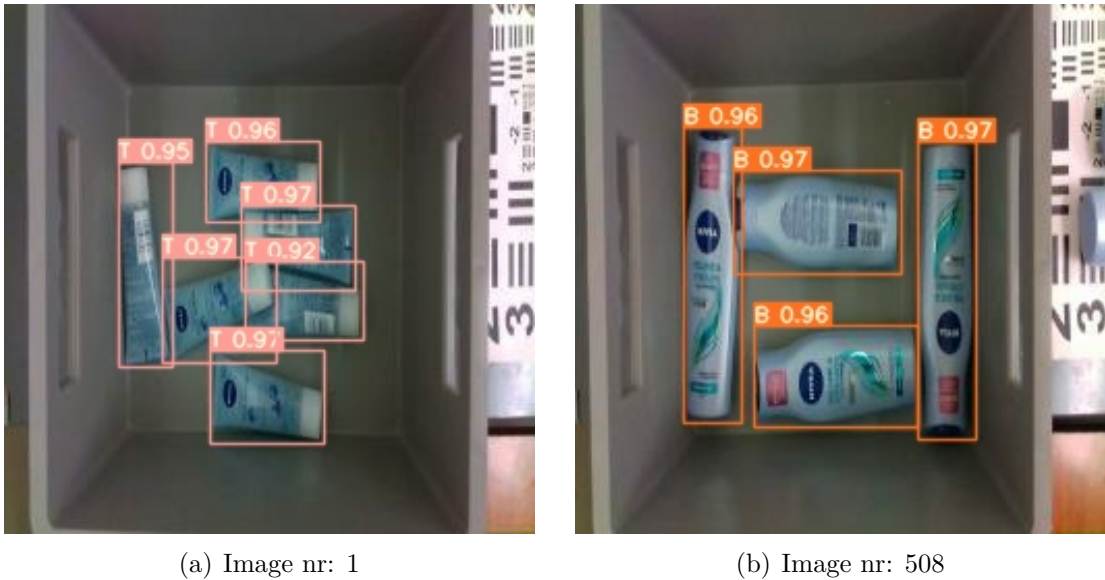


Figure 4.1: Examples from the detection run on the test part of the dataset.

The six objects in the second dataset that were not detected, was due to obstructions. An example of obstruction is shown in figure 4.2, where emphasis is put on the model not knowing the annotation of this image (from test dataset). Figure 4.2 (a) show one of the 16 images from the object detection run and (b) shows the same image illustrated in Roboflow [85] with annotations.

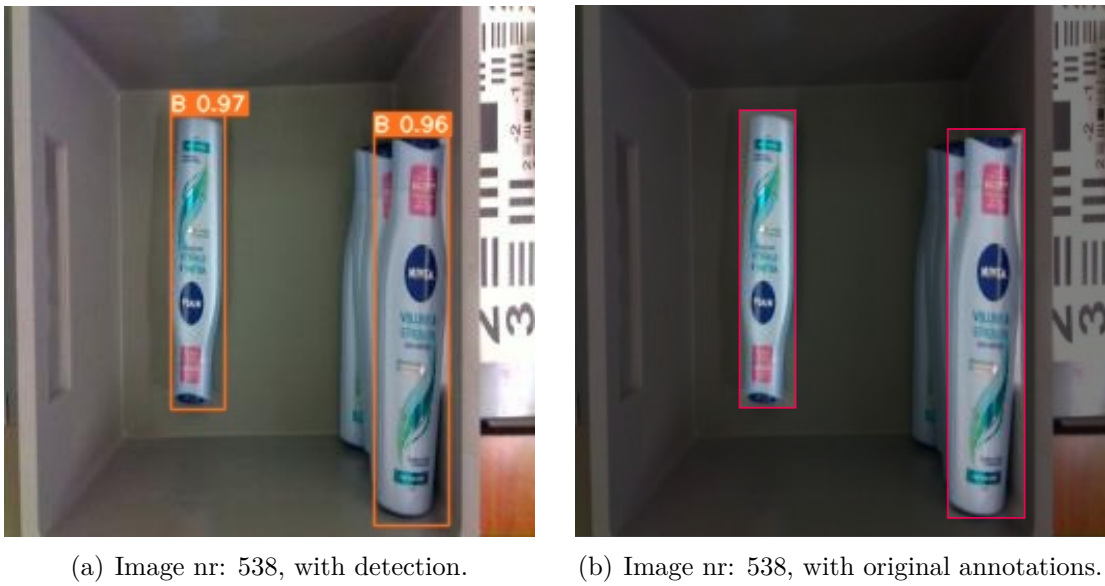


Figure 4.2: Examples that show how objects that are visible enough to be annotated are also being detected.

Result show a well performing object detector with a high detection rate and high certainty rate. Additionally, the figure 4.2 above show that the detector has made the bounding box in (a), because the coordinates are not completely the same as in (b).

## 4.2 Experiment: GAN generated objects

The purpose of this experiment is to find out how the GAN (DCGAN) perform on this type of dataset. The dataset consists of 151 images of one or more objects, of the same group, with different orientations (described as image group 1 in chapter 3.4.2).

### Experiment setup:

The experiment will be run on the local computer (chapter 3.1.1), in an empty memory folder (no pre-training). The batch size is set to 64, resulting in two iterations for every epoch. It is decided to train the discriminator more than the generator, by setting  $N_D$  value to 5. The model will be trained for a total of 100 000 epochs, 200 000 iterations for the discriminator and 40 000 for the generator. The training process will be sampled with 100 generated images, every 250 iteration for the generator, resulting in 161 images.

### Experiment results:

The training took  $\approx 44$  hours not accounting for rare but occurring crashes. These crashes were caused by accumulating RAM usage and revoked Windows permission, required for the program to write to the harddrive. The figure 4.3 show samples from regular intervals during the training, 5000 iteration apart starting at iteration 250. The event occurring from image (c) iteration 10 250 until image (d) 20 250, will be further elaborated below.

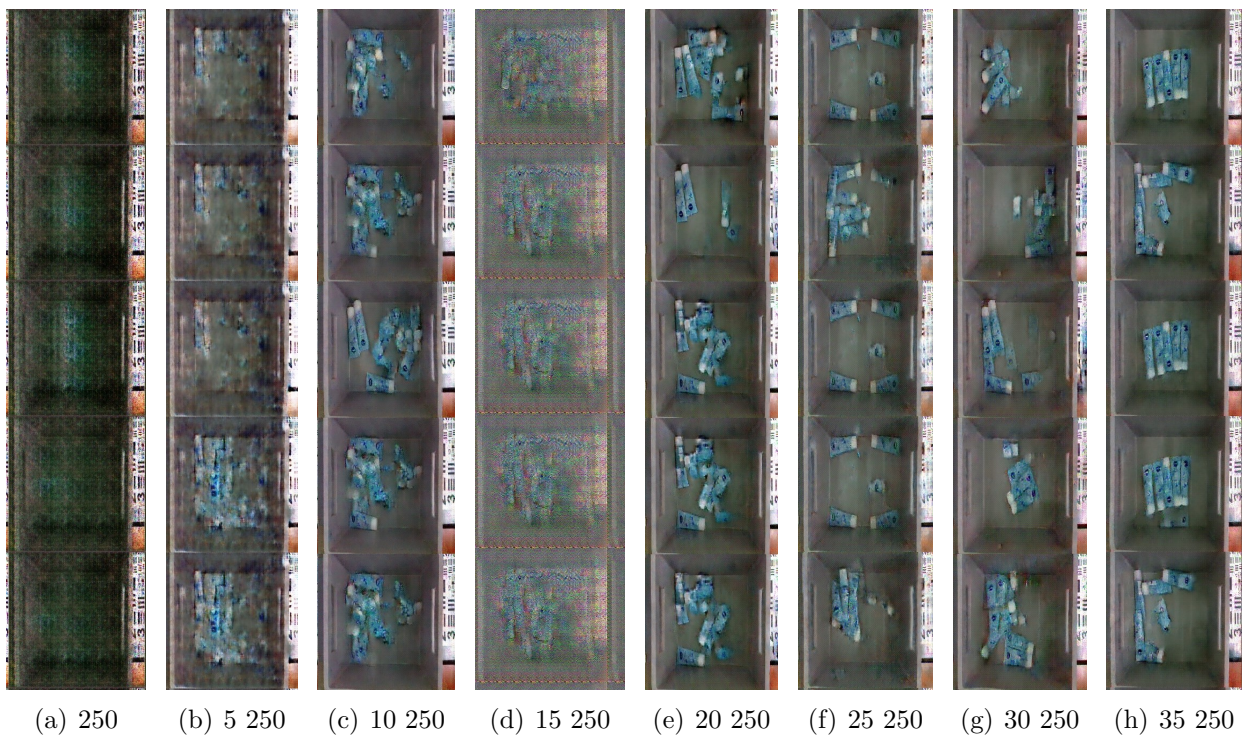


Figure 4.3: GAN training process. Showing 5 of 100 sampled images horizontally for every 5000-iteration interval of generator training. Images starting at iteration 250, since the iteration 0 image is just a gray noise image.

The expanded period around sample (d) from figure 4.3 are shown in figure 4.4. These samples illustrate oscillation effecting the model stability during training. Similar oscillations occur approximately every 5000 iteration, with inconsistent duration and severity. The example shown in figure 4.4 are the longest and most severe. The smallest oscillations are similar to (g) (in figure 4.4) for one single sample.

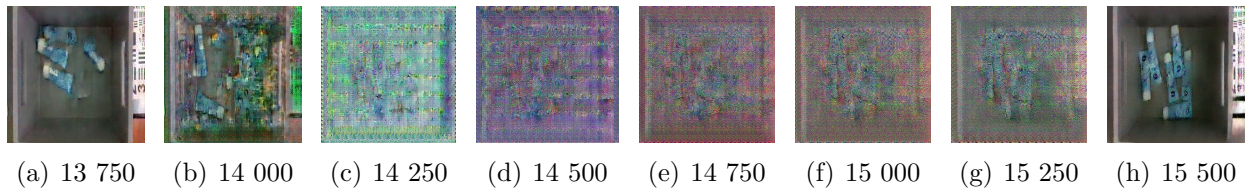


Figure 4.4: Oscillation during the GAN training process. Showing 1 of the 100 sampled images for iterations between 13 750 to 15 500 during generator training.

The last generated sample with good image quality is shown in figure 4.5. This is from iteration 39500, because both iteration 39750 and 40000 are affected by oscillations. There is mode collapse (when multiple generations result in the same creation), resulting in only 13 different images. From the 13 creations, three are identical to the dataset, in eight the objects are of too poor quality to be new generations and two show promise as newly generated images of low quality (example coordinates 1x9 and 10x10).

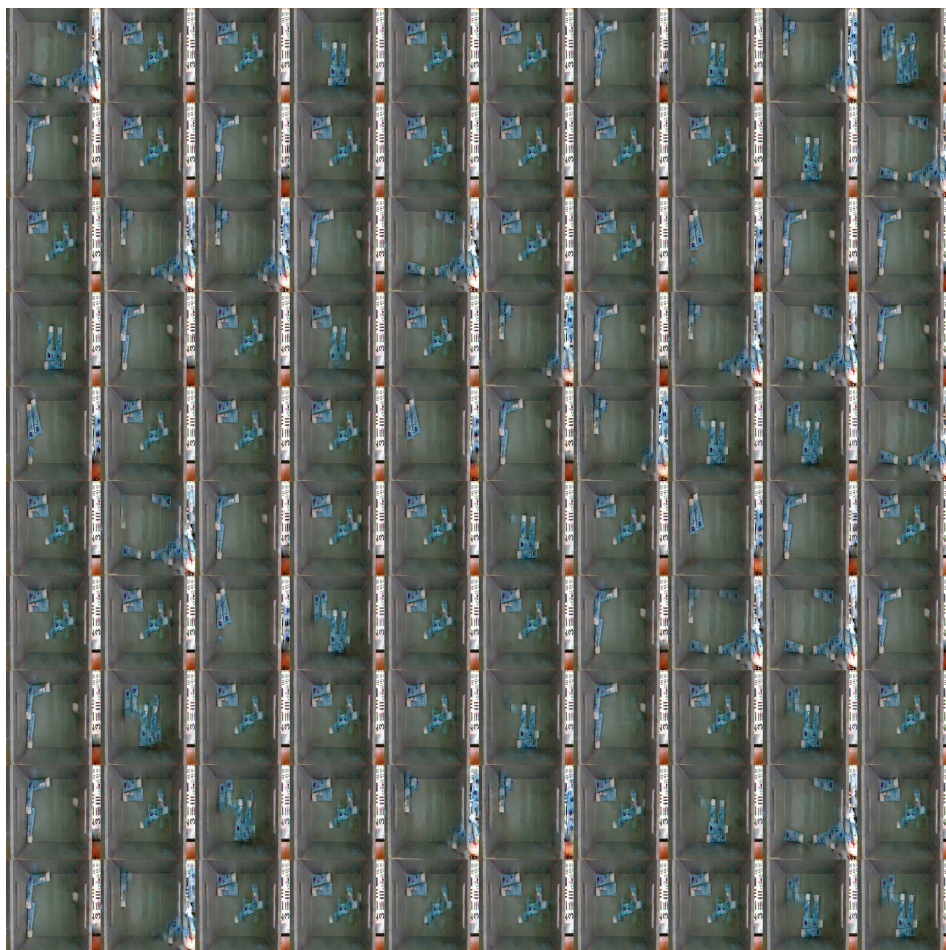


Figure 4.5: 100 generated images from iteration 39500. A larger version of this image can be found in appendix figure 1.

### 4.3 Experiment: WGAN-GP generated objects

The purpose of this experiment is to find out how the WGAN-GP perform on this type of dataset. The dataset consists of 151 images of one or more objects, of the same group, with different orientations (described as image group 1 in chapter 3.4.2).

#### Experiment setup:

The experiment will be run on the local computer (chapter 3.1.1), in an empty memory folder (no pre-training). The batch size is set to 64, resulting in two iterations for every epoch. It is decided to train the discriminator more than the generator, by setting  $N_D$  value to 5. The specific parameter for gradient penalty weight will be set at 10, as recommended in the original WGAN-GP paper [10]. The model will be trained for a total of 100 000 epochs, 200 000 iterations for the discriminator and 40 000 for the generator. The training process will be sampled with 100 generated images, every 250 iteration for the generator, resulting in 161 images.

#### Experiment results:

The training took  $\approx 70$  hours not accounting for rare but occurring crashes. The models were stable during training and continuously improving the quality of images generated, with a declining speed. There is no specific image version to continuously monitor during training, as the images are created at random. During training it is observed that the images that in general was containing fewer objects or objects with distance between them, was the generally best quality images.

Figure 4.6 visualize the models ability to improve the quality of the images, at specific intervals during training. The reason that it was possible to find this specific image at multiple intervals of training, is presumably because it is part of the training dataset and the original can be viewed in figure 3.7 in chapter 3.4.2.

The figure 4.6 does also visualize the difficulty evaluate the image quality progression the longer the model is training. The improvements from (a) to (c) is visible, from (c) to (d) is not so visible and from (d) to (f) becomes a matter of opinion. The same is true when inspecting and comparing all the 100 sample images at these intervals, it was therefore difficult to determine the progression after epoch 50 000 (5:1 ratio between discriminator and generator training, there are two iterations per epoch).

In the figure 4.7 are the 100 generated samples from the last iteration of this experiments, with some explanation below. The samples show one newly generated image, determined to be of acceptable quality. Additionally, samples from iteration 20 000 and 30 000 are in the appendix figures 2 and 3. The sample from iteration 20 000, show three newly generated images, determined to be of acceptable quality (the four newly generated images found during this experiment, will later be used for experiment 4.5 and shown in larger detail in figure 4.18).



(a) 5 000



(b) 10 000



(c) 16 000



(d) 20 000



(e) 29 750



(f) 40 000

Figure 4.6: Similar versions of one sample image during training, generator iteration listed below the image. A different but very similar image had to be used for iteration 10 000, because there was no similar image sample generated between iteration 9 000 and 16 000 (2800 sample images from 28 sample intervals).

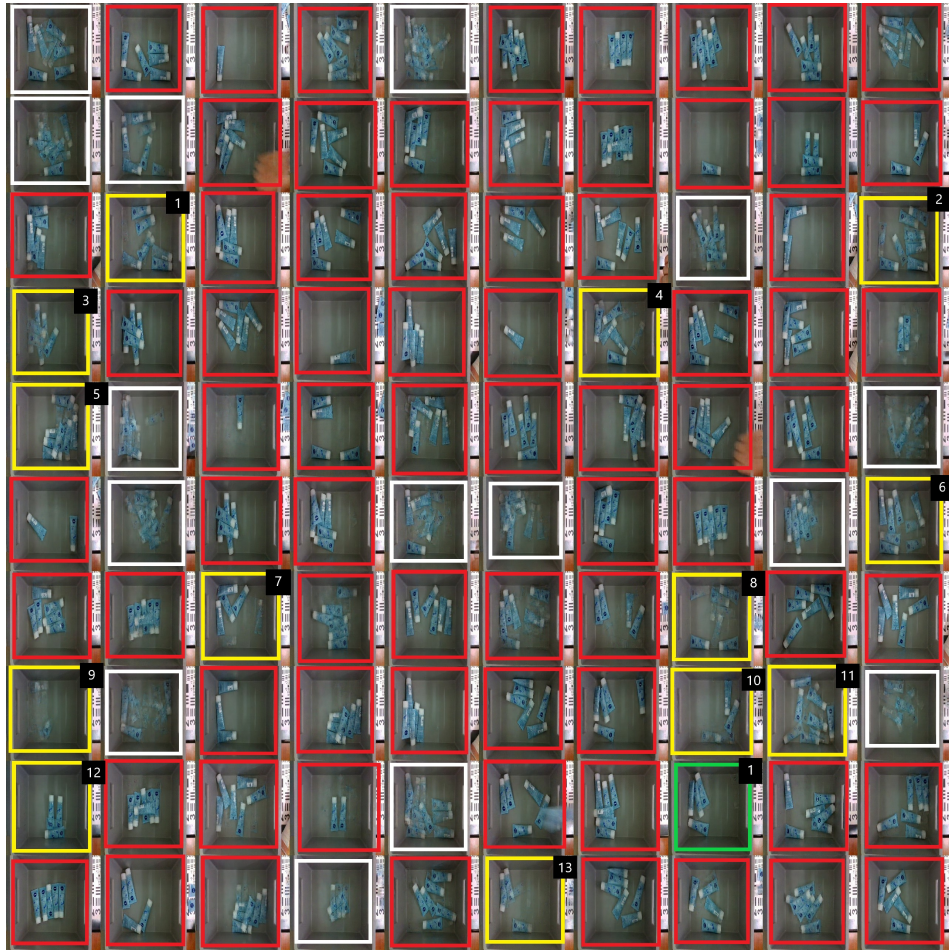


Figure 4.7: 100 Generated images from the model. White squares are too bad quality to be new generations and Red squares do already exist in the training dataset. The Yellow squares represent interesting generations and the Green squares that are new acceptable quality generations.

#### Yellow squares in figure 4.7:

1. Exist as 6 objects, but not as 7.
2. Poor quality, but there are more than 6 objects, therefore a new image (there are no images with more than 6 objects in the training dataset).
3. Do not exist as 6 objects, but is a blurry image.
4. Exist as 5 objects, do not exist as 6 objects.
5. Very bad quality, but more than 6 objects.
6. Poor quality, but there are more than 6 objects.
7. Does not exist as 4 objects, do exist as 5 objects.
8. Does not exist as 4, 5 or 6 objects, but have some blurriness.
9. Interesting that it is almost an empty box, because that does not exist in the dataset used for training.
10. Does not exist as 2 objects, do exist as 3.
11. Poor quality, but there are more objects than 6.
12. Exist as 3 and 4 objects, one object is almost removed.
13. New image, both as 3 and 4 objects, but not very clearly outlined objects.

#### Green squares in figure 4.7:

1. A good quality image of 4 objects that does not exist in the dataset.

## 4.4 Experiment: WGAN-GP with 10 images

The purpose of this experiment is to find out how the WGAN-GP perform with a dataset comprised of a limited amount of images. From the same dataset as used for the previous experiment, only ten of the 151 images are selected for this experiment, shown in figure 4.8.

### Experiment setup:

The experiment setup will be the same as for the previous experiment, with the following changes. The batch size is set to 8 (cannot be larger than the dataset size), resulting in one iteration for every epoch. The model will be trained for a total of 50 000 epochs, 50 000 iterations for the discriminator and 10 000 for the generator. The training process will be sampled with 100 generated images, every iteration for the generator for 2 000 iterations (of 10 000), resulting in 2001 images. After the initial 2 000 iterations, it will be sampled at intervals of 250, resulting in 33 images.

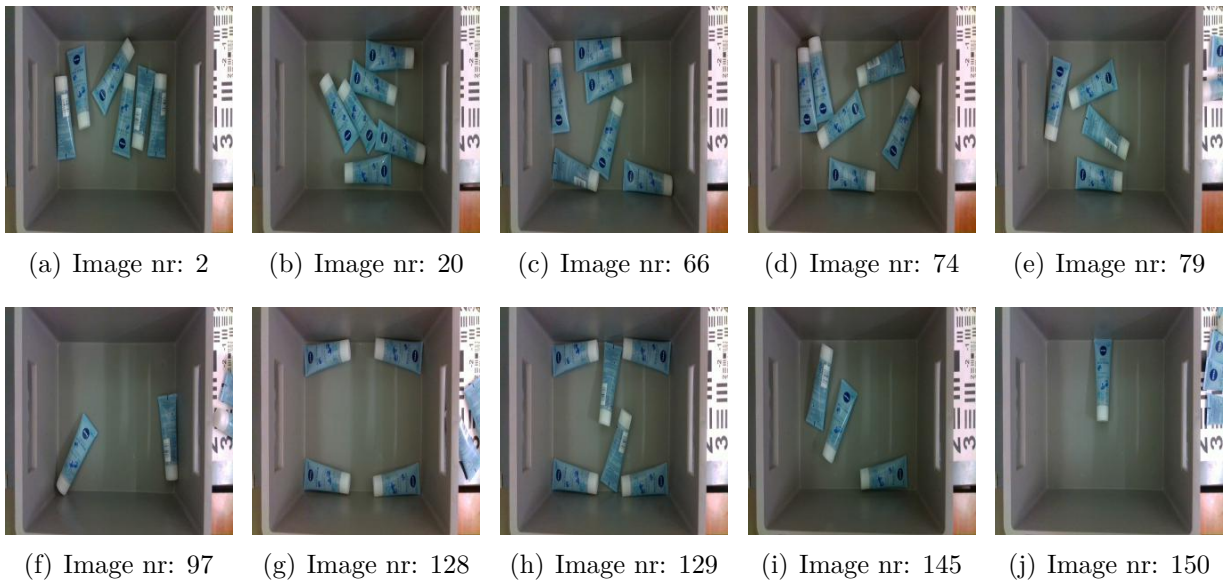


Figure 4.8: The reduced dataset of ten images that will be used in this experiment.

### Experiment results:

The training took  $\approx 17$  hours not accounting for rare but occurring crashes. The models were stable during training. The resulting samples showed less creativity than previous experiments, mostly learning the exact layout of the ten dataset images. There were still some newly generated images of sub-optimal quality created during the sampling, showed in figure 4.9.

Furthermore, the figure 4.9 shows that the generated images does not always improve. By examining (e) and (h), from iteration 10 000 and 7 250, it is the same generated image having lower quality 2 750 iterations later. Additionally, the figure show that there is randomness to when a good quality newly generated image will occur during training. This is shown by the generated images at different intervals in (g), (h) and (i), arguably being of better quality than most of (a) to (f).



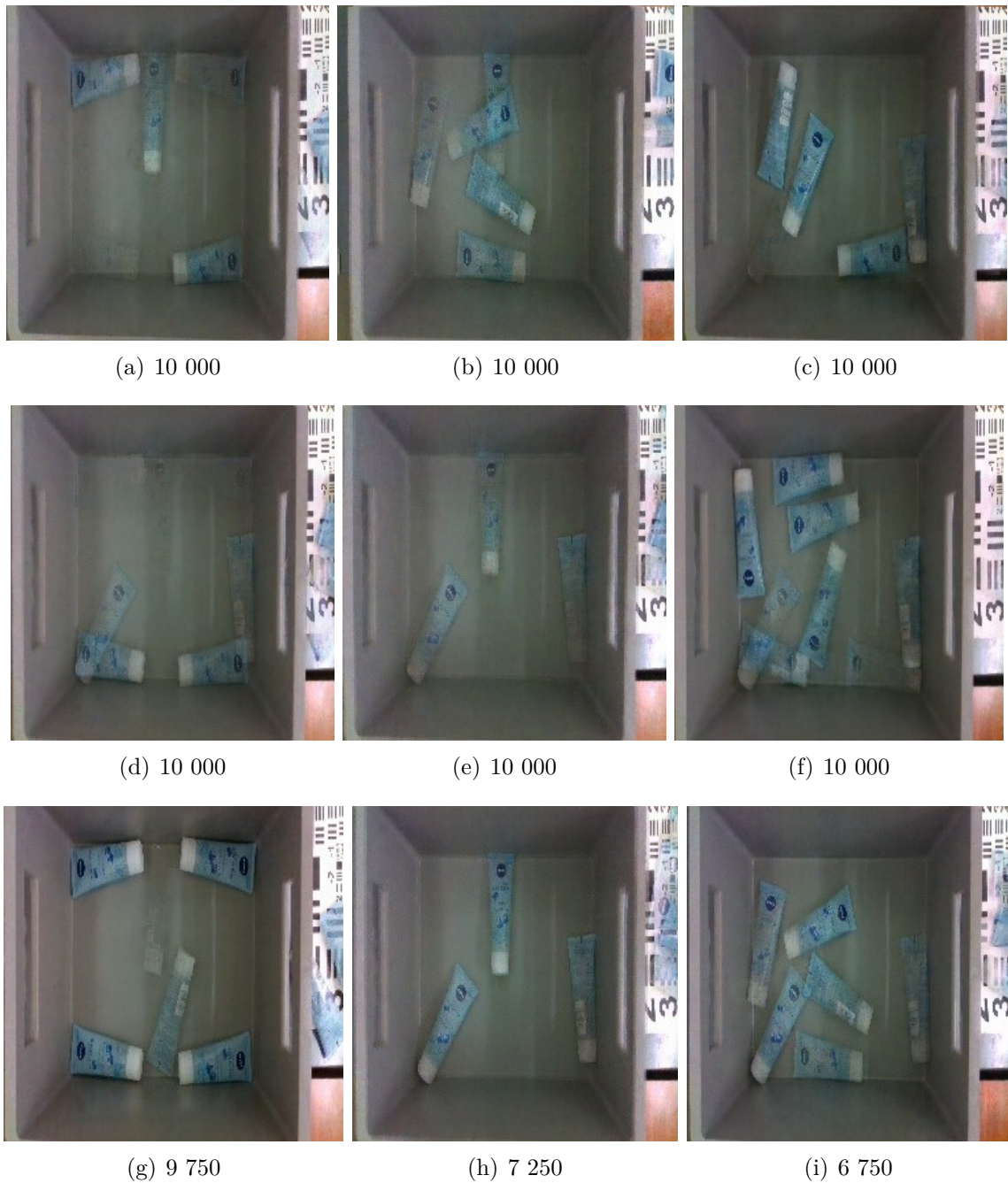


Figure 4.9: The best generated samples from the last iteration is show in figures (a-f), some good generations was observed in earlier samples and shown in figures (g-i). Figure (e) and (h) show a similar new generation, but the quality of the image has decreased with during the longer training.

### 4.4.1 Experiment: Testing pre-training

This experiment is to examine if a WGAN-GP may be pre-trained.

#### Experiment setup:

The setup will be the same as in experiment 4.4, with a small change. The dataset will be changed to ten images of another object and continue training from epoch 50 000. It will be trained for 10 000 epochs, resulting in 2 000 iterations for the generator. The resulting images will be compared with the images from experiment 4.4.

#### Experiment results:

The training took  $\approx 3$  hours not accounting for rare but occurring crashes. Figure 4.10 show samples for comparing normal training performed in experiment 4.4 with pre-training of a completely new dataset of ten unseen objects. A representative selection of the images generated after 1000 generator iterations are shown in 4.11. The result shows that pre-training is possible, furthermore it is seemingly giving an improved the training speed.

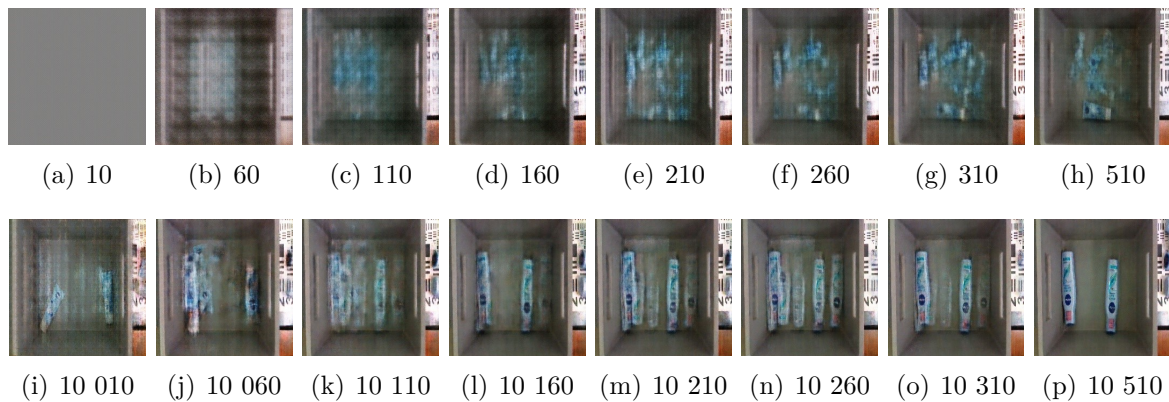


Figure 4.10: Generated samples from training of experiment 4.4 are shown in figures (a-h). Continued training of experiment 4.4 with ten new images as the dataset, are shown in figures (i-p). The sampled intervals are listed below the images.

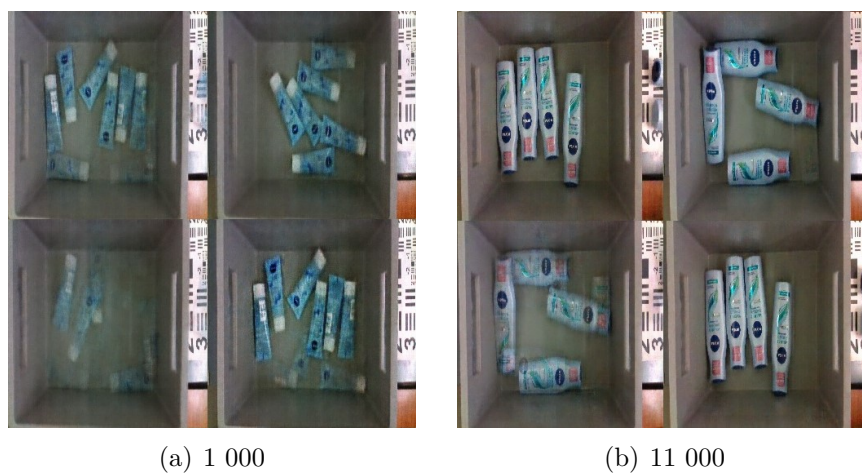


Figure 4.11: Representative samples of the 100 generated images after 1000 iterations of generator training.

## 4.4.2 Experiment: Pre-trained WGAN-GP with 10 images

This experiment is to examine how pre-training a WGAN-GP, before training it for the same amount as experiment 4.4 (50 000 epochs), will affect the generated images.

### Experiment pre-setup:

For this experiment the models need to be trained, so that it can serve as pre-trained models. Training will be performed with the same parameters as for experiment 4.4. All the images in the "RU Beiersdorf" dataset will be used for the pre-training, except images of mixed objects and the two object groups 1 and 4 (example of them are above in figure 4.11). The resulting dataset contains 1 801 images of twelve different object groups. The models are trained for 15 000 epochs, 3 375 000 discriminator iterations and 675 000 generator iterations. The limit was set at 15 000 epochs because of time constraints, using  $\approx 150$  hours to complete the training, not accounting for crashes. The training is sampled at intervals of 1 000 generator iterations, a portion of the last samples is shown in in figure 4.12.



Figure 4.12: 16 representative samples of the 100 created, to illustrate the model quality after 675 000 iterations. The whole sample image can be viewed in appendix figure 7.

### Experiment setup:

The experiment will be performed on the pre-trained model described above, changing the dataset of 1 801 images with the ten images from before (shown in figure 4.8). It will train for 50 000 more epochs, bringing the discriminator iterations to 3 425 000 and the generator iteration to 685 000. The sampling will be set at 250 generator iterations per sample.

### Experiment results:

The training took  $\approx 16$  hours. The best samples from the 100 generated images at the last iteration is shown in figure 4.13. The figure shows the ten best images, as they will be used for the next experiment.

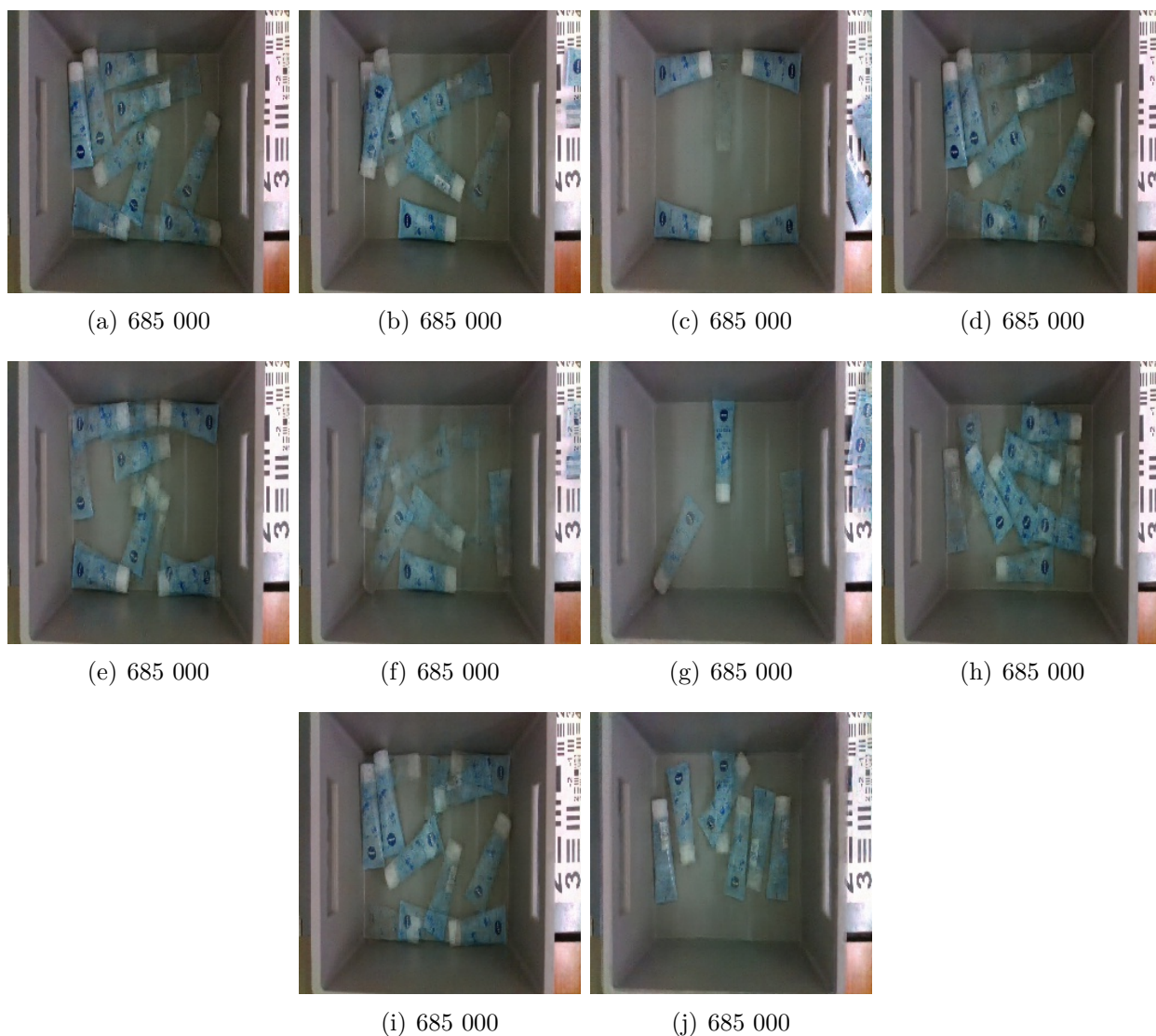


Figure 4.13: The best generated samples from the last iteration.

### 4.4.3 Experiment: Training with generated images in the dataset

This experiment is to examine if expanding the dataset with already generated images of sub-optimal quality, will help improve the quality of these images.

#### Experiment setup:

This experiment will be a continuation of experiment 4.4.2, with the images shown in figure 4.13 added to the dataset. Making the dataset a combination of ten real images and ten fake images, this will result in two iterations per epoch. The model will continue training for 50 000 epochs bringing the discriminator iterations to 3 475 000 and the generator iteration to 695 000. The sampling will be set at 250 generator iterations per sample.

#### Experiment results:

The training took  $\approx 16$  hours, not accounting for two crashes. One observation is that the model continued to perform one iteration per epoch instead of two.

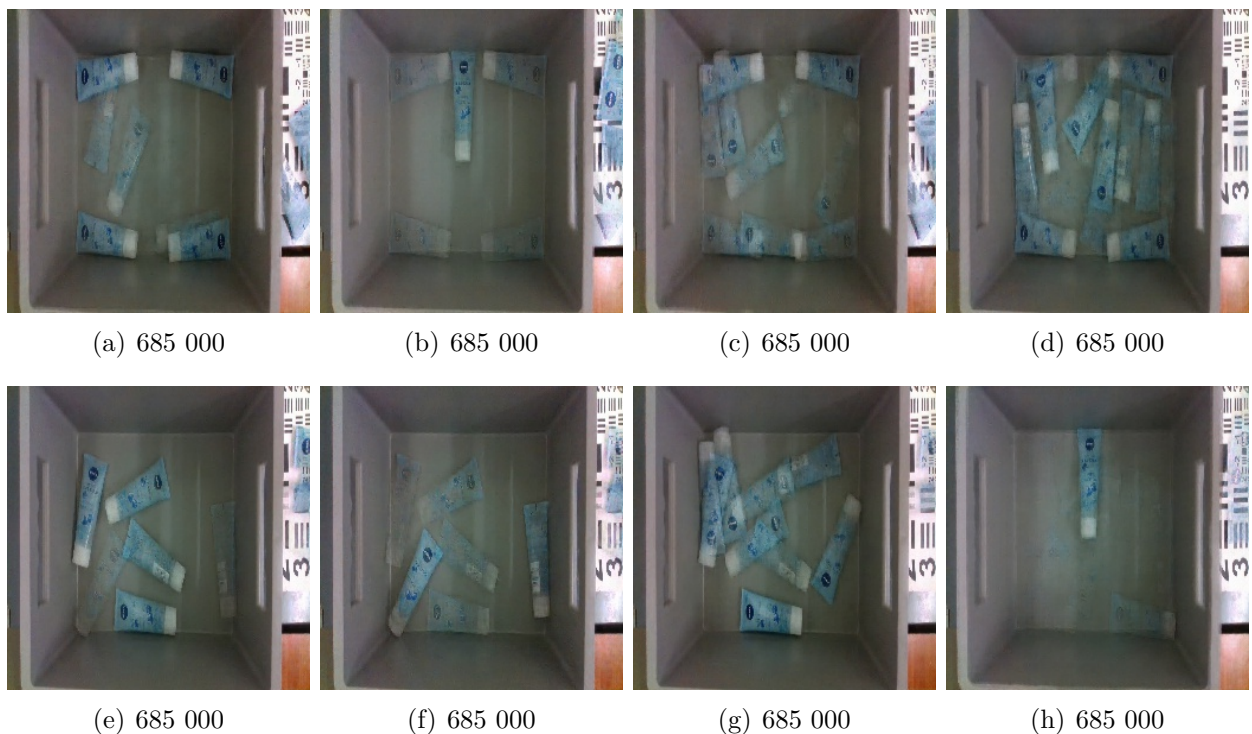


Figure 4.14: The best generated samples from the last iteration.

It was observed from the samples outputted during training, that there possibly were more newly generated images of improved quality than previously. The best samples are showed in figure 4.15 on next page.

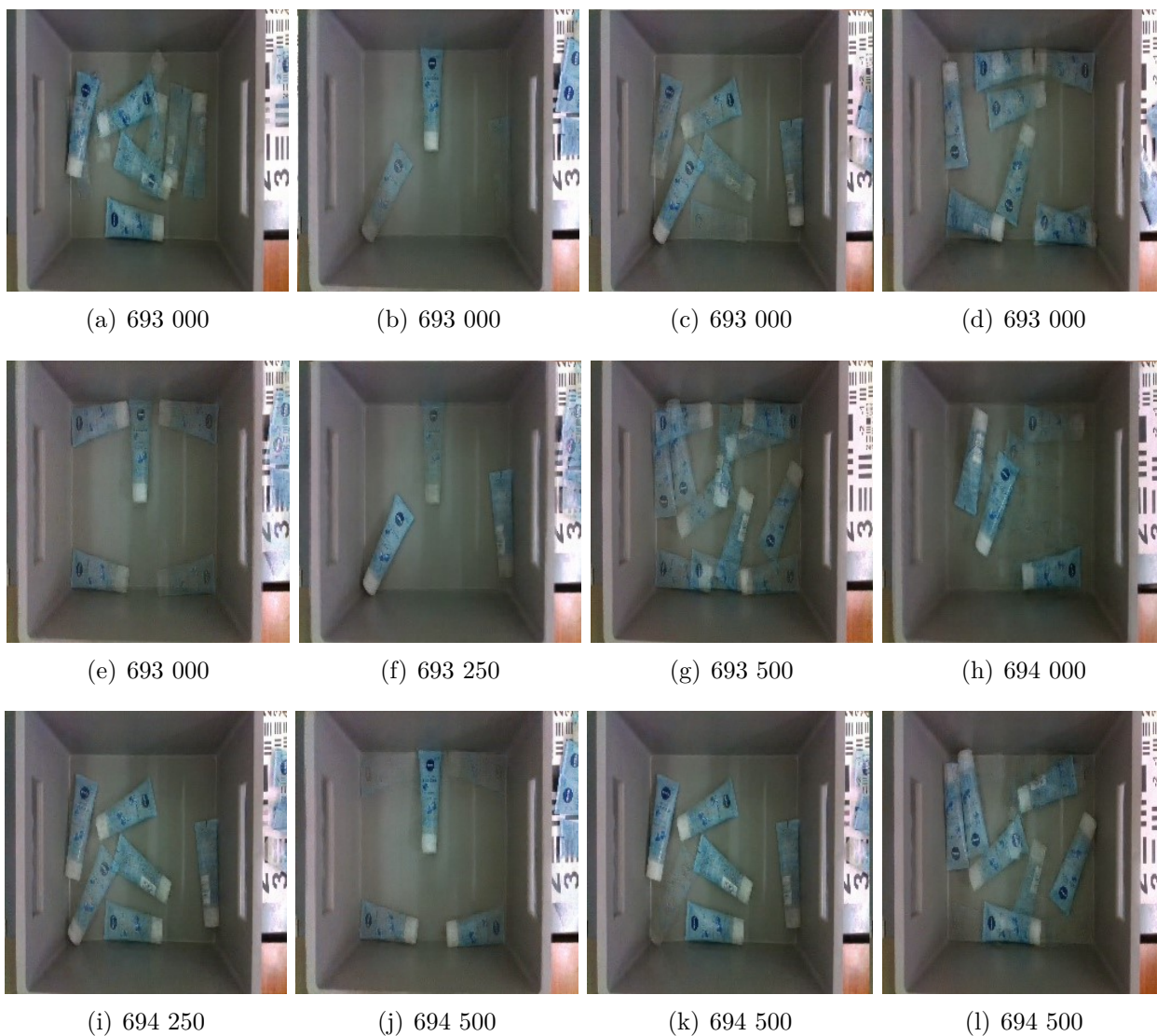


Figure 4.15: The best generated samples during training of the network. The sample iteration is listed below the images.

#### 4.4.4 Experiment: Expanding dataset with 10 images

The purpose of this experiment is to find out how the WGAN-GP perform with a larger dataset than the previous experiment. From the same dataset as used for the previous experiment (4.4), ten additional images are selected from the 151 images (the additional images are shown in figure 4.16) for this experiment (making the total images in this dataset twenty unique images).

##### Experiment setup:

The experiment setup will be the same as for the previous experiment 4.4, with the following changes. The batch size that is set at 8 (to be comparable with experiment 4.4), now results in two iterations for every epoch. The model will be trained for a total of 50 000 epochs, 100 000 iterations for the discriminator and 20 000 for the generator. The training process will be sampled with 100 generated images, every 250 iteration for the generator, resulting in 81 images.

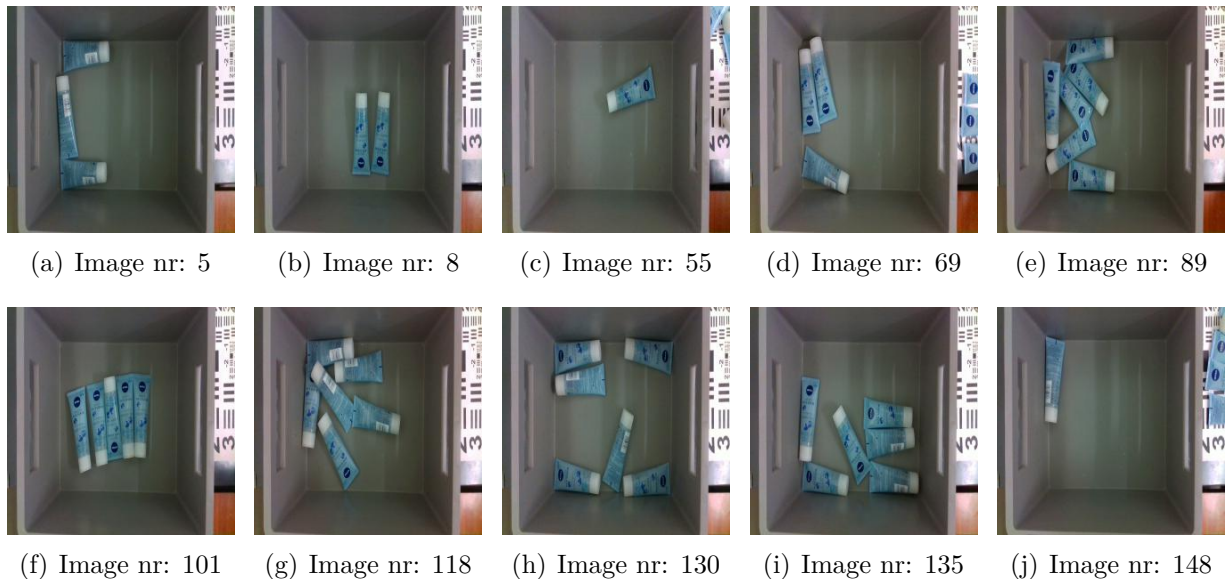


Figure 4.16: The additional ten images that will be added for this experiment.

##### Experiment results:

The training took  $\approx 37$  hours not accounting for rare but occurring crashes. The models were stable during training. The resulting images showed more creativity, as expected when the model has the possible combinations of images. The best newly generated images are shown in figure 4.17.

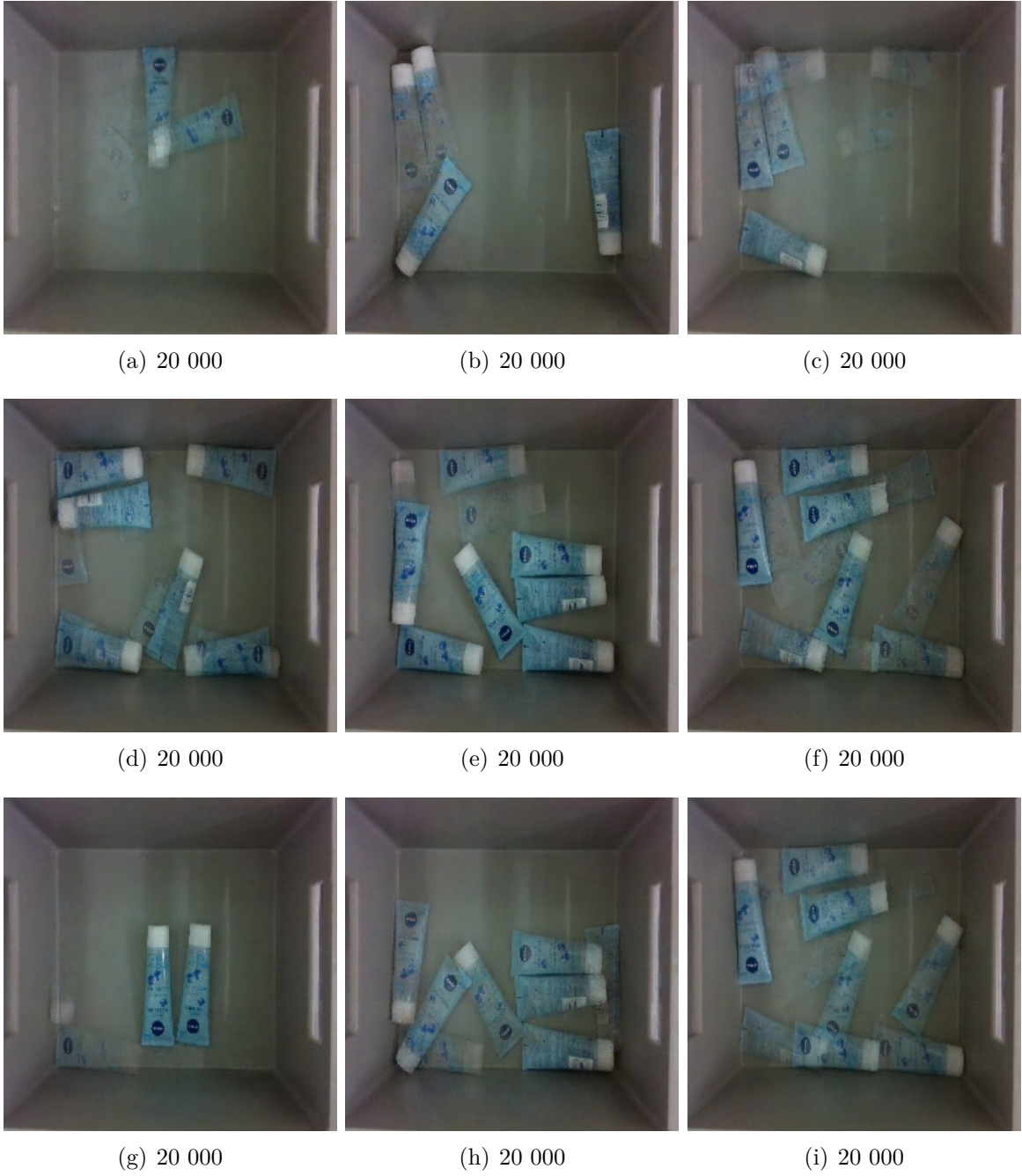


Figure 4.17: The best newly generated samples from the last iteration.



## 4.4.5 Comparing experiment results

To compare the results of the experiments performed in this sub-chapter, the 100 generated samples for 3 intervals are evaluated. The chosen intervals are spaced out by 1000 iterations from the last iteration counting backwards, this is to try and get a more impartial impression of the generations from the model. Each of the resulting 300 samples from each experiment is evaluated from the criteria listed below and presented as percentages in table 4.1.

- Green = New additions to the training dataset of sufficient quality.
- Yellow = Almost new additions to the training dataset of sub-optimal quality.
- White = Largely partial creation of objects.
- Red = Exactly similar to the training dataset.

Results of 300 samples evaluated from each experiment				
	Green	Yellow	White	Red
Experiment: 4.4	4%	9%	1.3%	85.7%
Experiment: 4.4.2	1.7%	7.7%	0.7%	90%
Experiment: 4.4.3	3%	6.7%	0.7%	89.7%
Experiment: 4.4.4	3.7%	15.7%	2%	78.7%

Table 4.1: The table is based on 300 samples from 3 intervals during training of each experiment. Each interval therefore consist of 100 generated samples distributed over the last 3000 iterations (for example iteration 8000, 9000 and 10000 are used for experiment 4.4)

All the images used for evaluations, can be found in the appendix.

## 4.5 Experiment: Detect generated objects with YOLOv5x

The (subjectively) best new images generated during the experiments are from experiments 4.3, 4.4 and 4.4.3. It would be of interest to confirm that it is possible to perform object detection on the images and that no unforeseen problems occur.

### Experiment setup:

Setup for this experiment is performed in experiment 4.1.

### Experiment results:

The detection with YOLOv5x took 11.34 seconds, largely detecting all the objects in the boxes with high certainty and showing no unforeseen problems with detection in the generated images. Results are shown in the figures below.

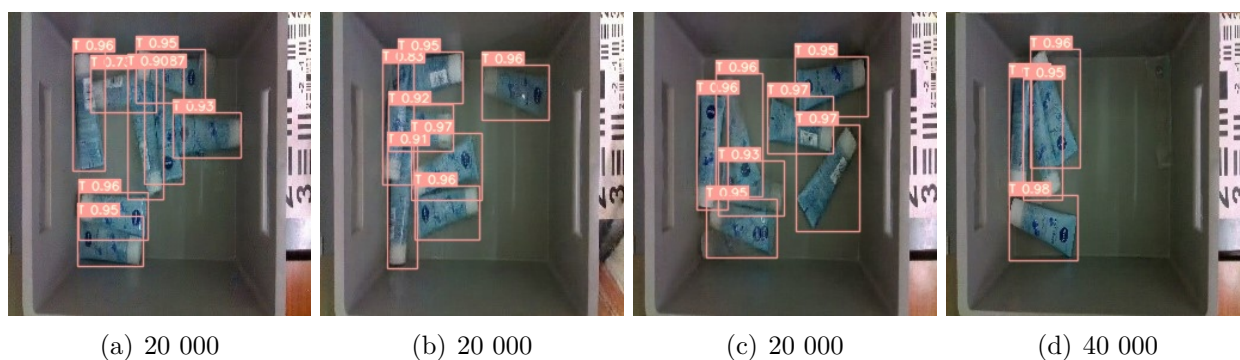


Figure 4.18: The best generated samples found during training in experiment 4.3.

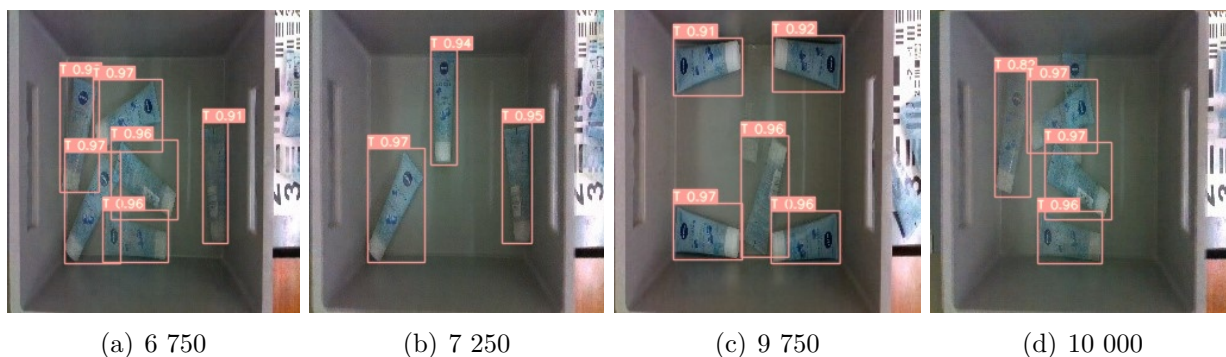


Figure 4.19: The best generated samples found during training in experiment 4.4.

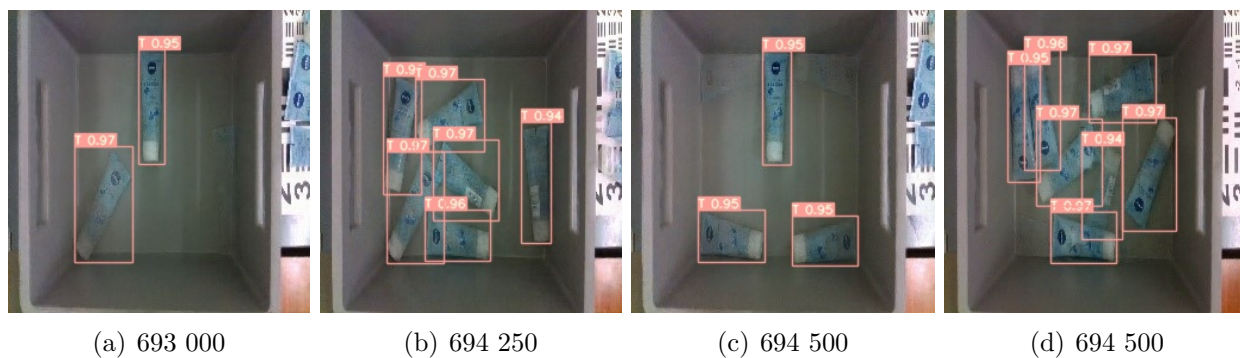


Figure 4.20: The best generated samples found during training in experiment 4.4.3.

# 5. Discussion and Future Work

In this chapter the results from the experiments performed in chapter 4 will be discussed, in addition to the suitability of using Generative Adversarial Networks as a solution to the proposed problem. Some suggestions on future work will also be explained.

## 5.1 Discussion about the experiments

### Evaluations performed:

Evaluating and choosing the best results from the experiments performed in this thesis, is done by manual inspection. Therefore, the decisions made may be very subjective and dependent on the person performing the manual inspections.

The resulting artificially generated images from the models contain (and are supposed to contain) large aspects of randomness. This randomness stems from the input of random noise to the generative model. The model is able to produce a wide variety of meaningful output images from sampling the random noise at different places for every image generated.

A consequence of this randomness is that the training process is difficult to follow, especially for the images wanted from the model, namely the new artificially generated (fake) images that are not part of the original training dataset. The reason is that the discriminator model will be penalized by the discriminator loss, for labeling a fake image as a real image. The loss function knows the origin of the images and will penalize all fake images that the discriminator labels as real. Thereby, only the images that are in the training dataset, where the generator model has learned the featured layout of the image, will stay in an equilibrium of continuously being generated. It is because these very similar images are labeled as both real and fake, when the discriminator loss corrects the weights of discriminator model (this is therefore feedback of little value, mentioned in chapter 2.3.3 paragraph three).

For the fake images that would be a new addition to the dataset, they will be reduced in quality or removed by this aforementioned process, this can be viewed in figure 4.9 visualized by the difference between (e) and (h). The generated samples similar to the ones in the dataset, seem to be able to visualize better the quality of objects produced by the model. This is because the fake images that would be an addition to the dataset, are evaluated as to have problems with transparency, not the quality of the objects (outline and details).

Evaluation with manual inspections are a comprehensive task for experimenting with Generative Adversarial Networks. Comparing the 300 generated sample images in experiment 4.3 with the 151 real images in the training dataset, results in 45 300 comparisons and only the finding of four images determined to be new additions of sufficient quality. The evaluation process gets easier with smaller datasets, making it possible to evaluate more generated images. Still the addition of an automatic method for selecting images or evaluate the quality of the generative model precisely, would provide a more comprehensive evaluation, probably resulting in the finding of better fake images or increasing the percentage of good fake images generated.

### **Experiment: GAN vs. WGAN-GP**

The experiment performed in 4.2 show limitations in the stability of the GAN, caused by oscillations as shown in figures 4.3 (c) and expanded upon in figure 4.4. The experiment also showed that the GAN suffers from mode collapse, where the generator model is only able to generate 13 different images in 100 generated samples, as shown in figure 4.5. The training of the GAN, did show that the GAN required less time training than the WGAN-GP in experiment 4.3, based on the same amount of epochs and the same amount of data, the GAN required  $\approx 50\%$  of the training time needed by the WGAN-GP. An unstable GAN that oscillates and are effected by mode collapse, does not constitute a good foundation for further experimentations, it does outweigh the benefits of the GAN, that are founded in simplicity of the architecture and the being less computationally demanding (less time consuming).

The results from the WGAN-GP experiment 4.3, did not show any oscillations. It was stable the whole duration of training for 40 000 epochs. In the figure 4.6 are similar generated images found during training, these similarly generated images are possible to find, because they are based on images in the training dataset and will therefore stay in equilibrium (do not ensure that it appears, just increases the probability). Comparing the images in the figure, gives an impression of the image quality generated, and it shows that it becomes difficult determining the quality improvements from epoch 20 000 (d) onwards to 40 000 (f).

The generated samples from iterations 20 000, 30 000 and 40 000, shown in figures 2, 3 and 4.7, did not suffer from the same mode collapse as the GAN. The inspection for sufficient quality generated images showed four promising generations, that are new additions to the training dataset (marked with green squares in the figures, 1.3% of the generated samples). Furthermore, it showed that generations similar to the training dataset made up 73.7% (red squares), generations of too poor quality made up 14.3% (white squares) and interesting almost new additions made up 10.7% (percentages are based all 300 samples from the three figures). The results show that the WGAN-GP is both stable and have the ability of creating new fake additions to the dataset, it will be more suitable than GAN, for use in further experiments.

### **Experiment: Generate images from a limited dataset**

The first experiment 4.4 on a limited amount of images, do expectantly, show less creativity than the previous experiment 4.3, as it has a smaller training dataset. The results are affected by this smaller dataset, because images containing fewer objects or objects with distance between them are chosen for this smaller training dataset, based on the experience from experiment 4.3. The smaller dataset will make it easier to evaluate the results and with less images containing many overlapping objects, the results should improve, as those images seemed to be the cause the images with poor quality and only partial object creations, in experiment 4.3.

The best generation of images from the last iteration are shown in figure 4.9 as (a to f), in addition there are added some images of good generations found during training in (g to i). The image of (e) and (h) do illustrate how images may disappear, caused by corrections made to the discriminator model.

Pre-training the WGAN-GP are proved to be possible in experiment 4.4.1, it does also show the possibility of improving the training speed between training datasets with images that has common environments (similar backgrounds).

Experimenting with a pre-trained model in 4.4.2 does show an increase of 4.3% in generations exactly similar to the training dataset (red) in table 4.1, but also a reduction of 0.6% in images with partial creations of objects (white). There might be three factors effecting this experiment, in an undesirable way. The first is that the model may be trained for a longer period of time than previously experiment 4.3, this being caused by improved training speed with pre-training (shown by experiment 4.4.1). The second is that the model quality is known to fluctuate, so the difference in result may be contributed by this fluctuation. The third is that to increase the effect pre-training might have, the ratio ( $N_D$ ) of training between generator and discriminator should possibly be turned down from 5:1.

The continued training of the model from experiment 4.4.2, after the addition of ten generated images, are performed in experiment 4.4.3. It gave an impression of improved results, that can be quantified by the comparison made in table 4.1. The table show that the model generated similar amount of images exactly similar to the training dataset (red, 0.3% difference) and exactly the same amount of images with partial creations of objects (white). Furthermore, the table does illustrate improvements made to the new additions of sufficient quality (green) by 1.3%, probably mostly at the behest of almost new additions of sub-optimal quality (yellow), that are reduced by 1%. Increasing green on the behest of mostly yellow is still an overall improvement. The improvement is presumably caused by the images being chosen from images that would have been categorized as yellow and green (images are shown in figure 4.13). The addition of these images to the dataset, should decrease the corrections on discriminator weights from generations similar to these images, as they will stay at equilibrium. The evidence is not strong enough state that the results are not caused by fluctuations in the model, but the distinct method applied makes the reason of fluctuations less likely.

In the experiment 4.4.4 the dataset is expanded with ten more real images shown in figure 4.16. The results from this experiment show more creativity, with a reduced percentage at 78.7% of generations exactly similar to the training dataset (red). This can also be confirmed by the number being even lower for experiment 4.3 at 73.7% (it must be stated that the intervals between these statistics are not the same, but the criteria for generations exactly similar to the dataset is the same).

### **Experiment: Object detector**

The experiments preformed with the object detector, proved their intended goal. Proving that fake images can be used with objection detection software. The software was simple to train as a tool in experiment 4.1 and resulted in a well performing object detector. For the experiment 4.5 at the end of chapter 4, it proved that the object detector determined the same amount of objects as determined by manual inspection with an average 94.4% certainty (three object certainties were not readable), the only exception was one obstructed objects in (d) figure 4.19. Obstructed objects are a known limitation for object detectors, mentioned in experiment 4.1. There were no unforeseen problems with using GAN generated fake images with object detector software (specifically the YOLOv5x software).

## 5.2 Discussion about the suitability

As a suitable method for problem proposed by Pickr.AI, the system based on WGAN-GP would reduce the need for manual effort with introducing new objects for picking. The WGAN-GP is able to generate more images from a limited dataset. Regardless, with the current evaluation methods for GANs it would just exchange one type of human labor with another, less physical. The WGAN-GP is not able to rotate and change the position of the objects, experiments shows only the possibility of combining objects from different images into the same image, thereby create a new addition. It does show the ability of splitting objects in the same image, then combining for a new image as preformed for image (b) in figure 4.15, a combination of (f) and (j) in figure 4.8.

Further explorations into modified GANs that allow for rotations and/or position change would make the solution more suitable. The development of new evaluation methods that give more concrete information about the model quality, would possibly increase the percentage of ideal generations. Additionally, an evaluation method that has the possibility of selecting generated images that are new additions to the training dataset of sufficient quality to be used, would provide a valuable tool for both application and experimentations with Generative Adversarial Networks.

## 5.3 Future work

The research field within generative models, where GAN only is a sub-field, might be able to solve many automation challenges in the future. The work performed in this thesis do only give an overview of solving one specific problem, using only one specific GAN version. There are still many advances being made within this research field, some of them might be able to solve the problem in this thesis better one day. Some methods that potentially would solve the problem better are listed below, along with development that would assist with the application and evaluation of GANs.

### 5.3.1 Deconstructing a full box of objects, with GAN inpainting

An interesting GAN method is proposed in "Context Encoders: Feature Learning by Inpainting" [109] in 2016. The paper uses GAN for something called inpainting, where the neural network is able to fill a hole made in an image. This or similar methods could have multiple interesting application, one would be the deconstruction of a full box of objects. This could work by using a precise object detector to find all the fully visible objects at the top of the box, cropping out the images within the bounding boxes created, then using the images for training. If the neural network is able to learn the feature layout of the objects correctly, it might be able to recreate the underlying objects (previously unseen). This would make a complete and new image. If this could be performed while containing the quality of the image, it should be possible to perform the operation for all the objects in the box, creating many different images from one single image.

### 5.3.2 Experimentation's preformed with new GAN versions.

There are many other GAN versions created, some are mentioned throughout this thesis. The evaluations between GAN versions, on the same datasets, have shown to resonate with human evaluations. Still the performance of different GAN versions is varied based on the type and size of the dataset, as previously mentioned in chapter 2.7. Giving rise to the belief that another GAN version might be able to solve this problem better, if tested. Another suggestion would also be to try the implementation of methods that may give the GAN possibility of rotating and changing the objects positions within the box.

### 5.3.3 Development of evaluation methods

Two apparent problems are that there currently are no evaluation methods that precisely predicts the quality of the generative model, and no automatic way of determining that a generated image is a new addition of great realistic quality. These tasks are required to be performed or at least confirmed with human labor, reducing the usefulness and effectiveness of using GAN methods to limit the need for human labor. Solving one or both of these problems would provide a valuable tool for the appliance of GAN methods for solving problems, for example like the one in this thesis.

## 6. Conclusion

This thesis had the purpose of determining if Generative Adversarial Networks would provide a suitable method for reducing the manual registration effort associated with the introducing of new objects for picking at an automatic robotic order fulfillment station. As a direct result the thesis also shows how a DCGAN and WGAN-GP perform on this uncustomary type of dataset.

Two interesting observations can be seen from the results. The first is that pre-training a WGAN-GP is possible and may contribute to faster training time, but it shows indications of decreasing the model quality. Secondly, adding generated images to the dataset during training shows indications of improving the quality of artificial images generated.

From the results in this thesis it can be concluded that the proposed solution is suitable for the problem but it has certain limitations. The WGAN-GP has the possibility of generating additional artificial images of sufficient quality, for the limited dataset in up to 4% of sample generations. These new additions are created by combining objects from different images, in a new image, but the objects retain their position and orientation in the image.

The limitations discovered, are that WGAN-GP is neither able to change the position nor the orientation of the objects, this limits the effectiveness of this solution. Additionally, the limitations of evaluation methods for model quality and image selection will result in evaluation by manual inspection. Manual inspection and the manual registration effort do both constitute as human labor, therefore will this method not currently reduce the human labor needed.



# Bibliography

- [1] S. Ytterstad, “Picking products from distribution containers by object detection and occlusion estimation,” *Master’s Thesis in Information Technology - Automation and Signal Processing - University of Stavanger*, June 2020.
- [2] I. Kandel and M. Castelli, “Transfer learning with convolutional neural networks for diabetic retinopathy image classification. a review,” *Applied Sciences*, vol. 10, p. 2021, 03 2020.
- [3] L. J. . J. ZHAO, “A survey on the new generation of deep learning in image processing,” *IEEE Access*, 2017.
- [4] A. Halthor, “Batch normalization - explained! - youtube,” [https://www.youtube.com/watch?v=DtEq44FTPM4&ab\\_channel=CodeEmporium](https://www.youtube.com/watch?v=DtEq44FTPM4&ab_channel=CodeEmporium), 2020 03, (Accessed on 05/10/2021).
- [5] U. Haputhanthri, “Introduction to deep convolutional generative adversarial networks using pytorch | by udith haputhanthri | the-ai.team | medium,” <https://medium.com/the-ai-team/introduction-to-deep-convolutional-generative-adversarial-networks-using-pytorch-92d88a19a574>, 05 2020, (Accessed on 05/22/2021).
- [6] Google, “The generator | generative adversarial networks | google developers,” <https://developers.google.com/machine-learning/gan/generator>, 2020, (Accessed on 05/15/2021).
- [7] J. Hui, “map (mean average precision) for object detection | by jonathan hui | medium,” [https://github.com/ultralytics/yolov5/blob/develop/README.md](https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-n-45c121a31173#:~:text=For%20COCO%2C%20AP%20is%20the,AP%40%5B.&text=95%5D%20corresponds%20to%20the%20average,on%2080%20categories%20(AP%40%5B., 03 2018, (Accessed on 05/27/2021).</li><li>[8] Ultralytics, “yolov5/readme.md at develop · ultralytics/yolov5 · github,” <a href=), (Accessed on 05/25/2021).
- [9] M. M. B. X. D. W.-F. S. O. A. C. Ian J. Goodfellow, Jean-Pouget Abadie and Y. Bengio, “Generative adversarial nets,” *Computer Science and Operational Research Department Montreal university*, 2014.
- [10] M. A. V. D. Ishaan Gulrajani, Faruk Ahmed and A. Courville, “Improved training of wasserstein gans,” *Montreal Institute for Learning Algorithms, Courant Institute of Mathematical Sciences and CIFAR Fellow*, December 2017.
- [11] J. Frankenfield, “Artificial intelligence (ai) definition,” [https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp#:~:text=Artificial%20intelligence%20\(AI\)%20](https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp#:~:text=Artificial%20intelligence%20(AI)%20)

- refers%20to,humans%20and%20mimic%20their%20actions.&text=The%20ideal%20characteristic%20of%20artificial,of%20achieving%20a%20specific%20goal., 03 2021, (Accessed on 06/01/2021).
- [12] “07313: Omsetning for varehandel, etter næring (sn2007) (mill. kr) 2008 - 2020. statistikkbanken. found by code 47.91 between the years of 2016-2020.” <https://www.ssb.no/statbank/table/07313/>, (Accessed on 05/05/2021).
- [13] BliVakker, “Om blivakker.no | gratis frakt - rask levering,” <https://www.blivakker.no/kundesenter-om-oss>, (Accessed on 06/01/2021).
- [14] PICKR.AI, “Pickr.ai,” <https://www.pickr.ai/>, (Accessed on 01/20/2021).
- [15] M. Hargrave, “Deep learning definition,” <https://www.investopedia.com/terms/d/deep-learning.asp#:~:text=Deep%20learning%20is%20a%20subset,learning%20or%20deep%20neural%20network.>, 2021, (Accessed on 05/30/2021).
- [16] S. Bjarnason, “Detecting previously unseen objects with-out human intervention using neural networks.” *Master’s Thesis in Mechatronics Engineering - School of Science and Engineering - University Reykjavik*, January 2021.
- [17] J. Natten, “Generative adversarial networks for improving face classification,” *Master’s Thesis - University of Agder*, 2017.
- [18] I. Dabbura, “Coding neural network — forward propagation and backpropagation | by imad dabbura | towards data science,” <https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagation-ccf8cf369f76>, 04 2018, (Accessed on 05/30/2021).
- [19] R. K. Samer Hijazi and C. Rowen, “cnn\_wp.pdf,” [https://ip.cadence.com/uploads/901/cnn\\_wp-pdf#:~:text=CNNS%20are%20used%20in%20variety,feature%20extractors%20are%20hand%20designed.](https://ip.cadence.com/uploads/901/cnn_wp-pdf#:~:text=CNNS%20are%20used%20in%20variety,feature%20extractors%20are%20hand%20designed.), (Accessed on 04/21/2021).
- [20] J. Brownlee, “How do convolutional layers work in deep learning neural networks?” <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>, 2020 04, (Accessed on 04/22/2021).
- [21] —, “A gentle introduction to padding and stride for convolutional neural networks,” <https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/>, 08 2019, (Accessed on 04/23/2021).
- [22] —, “Crash course in convolutional neural networks for machine learning,” <https://machinelearningmastery.com/crash-course-convolutional-neural-networks/>, 08 2019, (Accessed on 04/23/2021).
- [23] A. Anwar, “What is transposed convolutional layer? | by aqeel anwar | towards data science,” <https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11>, 03 2020, (Accessed on 05/10/2021).
- [24] H. Kumar, “Autoencoder: Downsampling and upsampling,” <https://kharshit.github.io/blog/2019/02/15/autoencoder-downsampling-and-upsampling>, 02 2019, (Accessed on 05/10/2021).

- [25] S. Choudhary, “How to choose best activation function for you model | by siddharth choudhary | medium,” <https://medium.com/@siddharth654choudhary/how-to-choose-best-activation-function-for-you-model-8af90557245b>, 11 2019, (Accessed on 05/11/2021).
- [26] H. S, “Activation functions : Sigmoid, relu, leaky relu and softmax basics for neural networks and deep learning | by himanshu s | medium,” <https://medium.com/@himanshuxd/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e#:~:text=ReLU%20is%20non%2Dlinear%20and,to%20the%20antisymmetry%20of%20tanh.>, 01 2019, (Accessed on 05/11/2021).
- [27] J. Brownlee, “How to choose an activation function for deep learning,” <https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/>, 01 2021, (Accessed on 04/28/2021).
- [28] D. Becker, “Rectified linear units (relu) in deep learning | kaggle,” <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning>, 2018, (Accessed on 04/28/2021).
- [29] J. Brownlee, “A gentle introduction to pooling layers for convolutional neural networks,” <https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/>, 07 2019, (Accessed on 04/24/2021).
- [30] —, “Tips for training stable generative adversarial networks,” <https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/#:~:text=In%20GANs%2C%20the%20recommendation%20is,downsampling%20in%20the%20discriminator%20model.>, 09 2019, (Accessed on 04/24/2021).
- [31] S. C. Alec Radford, Luke Metz, “[1511.06434] unsupervised representation learning with deep convolutional generative adversarial networks,” <https://arxiv.org/abs/1511.06434>, 01 2016, (Accessed on 06/03/2021).
- [32] S. Saxena, “Batch normalization | what is batch normalization in deep learning,” <https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/>, 03 2021, (Accessed on 05/10/2021).
- [33] S. Ioffe and C. Szegedy, “[1502.03167] batch normalization: Accelerating deep network training by reducing internal covariate shift,” <https://arxiv.org/abs/1502.03167>, 03 2015, (Accessed on 05/10/2021).
- [34] A. I. Shibani Santurkar, Dimitris Tsipras and A. Madry, “[1805.11604] how does batch normalization help optimization?” <https://arxiv.org/abs/1805.11604>, 04 2019, (Accessed on 05/10/2021).
- [35] Y. S. C.-Y. H. B. L. Guangyong Chen, Pengfei Chen and S. Zhang, “[1905.05928] rethinking the usage of batch normalization and dropout in the training of deep neural networks,” <https://arxiv.org/abs/1905.05928>, 05 2019, (Accessed on 05/10/2021).
- [36] F. Kratzert, “Understanding the backward pass through batch normalization layer,” <https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>, 02 2016, (Accessed on 05/10/2021).

- [37] DeepLearning.AI, “Normalizing inputs - practical aspects of deep learning | coursera,” <https://www.coursera.org/lecture/deep-neural-network/normalizing-inputs-lXv6U>, (Accessed on 05/10/2021).
- [38] N. Vijayrania, “Different normalization layers in deep learning | by Nilesh Vijayrania | towards data science,” <https://towardsdatascience.com/different-normalization-layer-s-in-deep-learning-1a7214ff71d6>, 12 2020, (Accessed on 05/10/2021).
- [39] Z. Jost, “Overview of GANs (generative adversarial networks) - part i | by Zak Jost | towards data science,” <https://towardsdatascience.com/overview-of-gans-generative-adversarial-networks-part-i-ac78ec775e31>, 10 2017, (Accessed on 04/07/2021).
- [40] N.-N. IITM, “Deep learning part - ii (cs7015): Lec 22.2 generative adversarial networks - architecture - youtube,” [https://www.youtube.com/watch?v=uyuYfTMHZM0&a\\_b\\_channel=NPTEL-NOCIITM](https://www.youtube.com/watch?v=uyuYfTMHZM0&a_b_channel=NPTEL-NOCIITM), (Accessed on 05/17/2021).
- [41] Techopedia, “What is an input layer? - definition from techopedia,” <https://www.techopedia.com/definition/33262/input-layer-neural-networks#:~:text=The%20input%20layer%20of%20a,for%20the%20artificial%20neural%20network>, (Accessed on 05/15/2021).
- [42] C. Lazarou, “Why do GANs need so much noise?. visualizing how GANs learn in... | by Conor Lazarou | towards data science,” <https://towardsdatascience.com/why-do-gans-need-so-much-noise-1eae6c0fb177>, 02 2020, (Accessed on 05/21/2021).
- [43] J. Brownlee, “How to explore the GAN latent space when generating faces,” <https://machinelearningmastery.com/how-to-interpolate-and-perform-vector-arithmetic-with-faces-using-a-generative-adversarial-network/>, 11 2020, (Accessed on 05/21/2021).
- [44] Google, “Overview of GAN structure | generative adversarial networks,” [https://developers.google.com/machine-learning/gan/gan\\_structure](https://developers.google.com/machine-learning/gan/gan_structure), 2020, (Accessed on 05/15/2021).
- [45] —, “The discriminator | generative adversarial networks,” <https://developers.google.com/machine-learning/gan/discriminator>, 2020, (Accessed on 05/18/2021).
- [46] —, “GAN training | generative adversarial networks | Google Developers,” <https://developers.google.com/machine-learning/gan/training>, 2020, (Accessed on 05/18/2021).
- [47] J. Brownlee, “Loss and loss functions for training deep learning neural networks,” <https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/>, 10 2019, (Accessed on 05/15/2021).
- [48] —, “Gentle introduction to the Adam optimization algorithm for deep learning,” <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>, 01 2021, (Accessed on 05/16/2021).
- [49] M. Peixeiro, “The 3 best optimization methods in neural networks | by Marco Peixeiro | towards data science,” <https://towardsdatascience.com/the-3-best-optimization-methods-in-neural-networks-40879c887873>, 03 2019, (Accessed on 05/16/2021).

- [50] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” <https://arxiv.org/pdf/1412.6980.pdf>, 01 2017, (Accessed on 05/25/2021).
- [51] B. Ghosh, I. Dutta, A. Carlson, M. Totaro, and M. Bayoumi, “An empirical analysis of generative adversarial network training times with varying batch sizes,” [https://www.researchgate.net/publication/344544069\\_An\\_Empirical\\_Analysis\\_of\\_Generative\\_Adversarial\\_Network\\_Training\\_Times\\_with\\_Varying\\_Batch\\_Sizes](https://www.researchgate.net/publication/344544069_An_Empirical_Analysis_of_Generative_Adversarial_Network_Training_Times_with_Varying_Batch_Sizes), 10 2020, (Accessed on 05/25/2021).
- [52] J. Brownlee, *Generative Adversarial Networks with Python - Deep Learning Generative Models for Image Synthesis and Image Translation (Edition: v1.5)*. Machine Learning Mastery, 2019, p. 60.
- [53] J. Hui, “Gan — ways to improve gan performance | by jonathan hui | towards data science,” <https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b>, 06 2018, (Accessed on 05/25/2021).
- [54] M. Jayathilaka, “Understanding and optimizing gans (going back to first principles) | by mirantha jayathilaka | towards data science,” <https://towardsdatascience.com/understanding-and-optimizing-gans-going-back-to-first-principles-e5df8835ae18>, 03 2018, (Accessed on 05/25/2021).
- [55] I. Gemp and B. McWilliams, “The unreasonable effectiveness of adam on cycles,” <https://sgo-workshop.github.io/CameraReady2019/11.pdf>, (Accessed on 05/25/2021).
- [56] J. Brownlee, “How to evaluate generative adversarial networks,” <https://machinelearningmastery.com/how-to-evaluate-generative-adversarial-networks/>, 08 2019, (Accessed on 05/02/2021).
- [57] Google, “Loss functions | generative adversarial networks | google developers,” <https://developers.google.com/machine-learning/gan/loss>, 2020, (Accessed on 05/18/2021).
- [58] PyTorch, “Dcgan tutorial — pytorch tutorials 1.7.1 documentation,” [https://pytorch.org/tutorials/beginner/dcgan\\_faces\\_tutorial.html?fbclid=IwAR2jeiIDv7MjYnI0xIEH01eytxVcxvMyD4oYnx3qDymUB5Ql2zeDZNN26FQ](https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html?fbclid=IwAR2jeiIDv7MjYnI0xIEH01eytxVcxvMyD4oYnx3qDymUB5Ql2zeDZNN26FQ), 2021, (Accessed on 02/01/2021).
- [59] D. Foster, *Generative Deep Learning - Teaching Machines to Paint, Write, Compose and Play*. O’Reilly, July 2019, pp. 97–127.
- [60] J. Hui, “Gan — wasserstein gan & wgan-gp. training gan is hard. models may never... | by jonathan hui | medium,” <https://jonathan-hui.medium.com/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490>, 06 2018, (Accessed on 05/19/2021).
- [61] L. Weng, “From gan to wgan, (lilianweng.github.io/lil-log),” <https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html>, 2017, (Accessed on 05/19/2021).
- [62] S. C. Martin Arjovsky and L. Bottou, “Wasserstein gan,” *Courant Institute of Mathematical Sciences and Facebook AI Research*, 2017.
- [63] J. Hui, “Gan — spectral normalization. gan is vulnerable to mode collapse and... | by jonathan hui | medium,” <https://jonathan-hui.medium.com/gan-spectral-normalization-893b6a4e8f53>, 02 2020, (Accessed on 05/19/2021).

- [64] ProgrammerSought, “Introduction to wgan-gp (improved wgan) - programmer sought,” <https://www.programmersought.com/article/78351384771/>, (Accessed on 05/19/2021).
- [65] J. Brownlee, “A tour of generative adversarial network models,” <https://machinelearningmastery.com/tour-of-generative-adversarial-network-models/>, 07 2019, (Accessed on 06/03/2021).
- [66] H. L. S. Z. X. W. X. H.-D. M. Han Zhang, Tao Xu, “[1612.03242] stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks,” <https://arxiv.org/abs/1612.03242>, 12 2016, (Accessed on 06/03/2021).
- [67] S. O. Mehdi Mirza, “[1411.1784] conditional generative adversarial nets,” <https://arxiv.org/abs/1411.1784>, 11 2014, (Accessed on 06/03/2021).
- [68] P. I. A. A. E. Jun-Yan Zhu, Taesung Park, “[1703.10593] unpaired image-to-image translation using cycle-consistent adversarial networks,” <https://arxiv.org/abs/1703.10593>, 03 2017, (Accessed on 06/03/2021).
- [69] TensorFlow, “Tensorflow,” <https://www.tensorflow.org/>, (Accessed on 04/06/2021).
- [70] PyTorch, “Pytorch,” <https://pytorch.org/>, (Accessed on 06/03/2021).
- [71] Python, “Welcome to python.org,” <https://www.python.org/>, (Accessed on 06/03/2021).
- [72] A. Borji, “[1802.03446] pros and cons of gan evaluation measures,” <https://arxiv.org/abs/1802.03446>, 10 2018, (Accessed on 05/02/2021).
- [73] M. M. S. G. O. B. Mario Lucic, Karol Kurach, “[1711.10337] are gans created equal? a large-scale study,” <https://arxiv.org/abs/1711.10337>, 10 2018, (Accessed on 06/03/2021).
- [74] A. Vidhya, “Datasets for deep learning | open datasets for deep learning,” <https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/>, (Accessed on 06/03/2021).
- [75] J. Brooks, “Coco annotator,” <https://github.com/jsbroks/coco-annotator>, 2019, (Accessed on 06/02/2021).
- [76] NVIDIA, “Cuda gpus | nvidia developer,” <https://developer.nvidia.com/cuda-gpus>, (Accessed on 04/05/2021).
- [77] TensorFlow, “Gpu support | tensorflow,” <https://www.tensorflow.org/install/gpu>, (Accessed on 04/05/2021).
- [78] Google, “Colaboratory – google,” <https://research.google.com/colaboratory/faq.html>, (Accessed on 04/09/2021).
- [79] A. G. C, “Setting up your gpu for tensorflow 2.4 (2021) | by alejandro granados c | medium,” <https://alejandro-gc.medium.com/setting-up-your-gpu-for-tensorflow-2-4-2021-d98cac79a686>, 01 2021, (Accessed on 04/05/2021).

- [80] A. Watson, “Tensorflow 2.4 with gpu in 4 steps | towards data science,” <https://towardsdatascience.com/install-tensorflow-with-cuda-cudnn-and-gpu-support-in-4-easy-steps-954f176daac3>, 01 2021, (Accessed on 04/05/2021).
- [81] TensorFlow, “Install tensorflow 2,” <https://www.tensorflow.org/install>, (Accessed on 04/06/2021).
- [82] J. Kitson, “Installing tensorflow with cuda, cudnn and gpu support on windows 10 | by dr. joanne kitson | towards data science,” <https://towardsdatascience.com/installing-tensorflow-with-cuda-cudnn-and-gpu-support-on-windows-10-60693e46e781>, 04 2019, (Accessed on 04/05/2021).
- [83] “python - difference between installation libraries of tensorflow gpu vs cpu - stack overflow,” <https://stackoverflow.com/questions/52624703/difference-between-installation-libraries-of-tensorflow-gpu-vs-cpu>, (Accessed on 04/09/2021).
- [84] S. Bjarnason, “Ru beiersdorf dataset,” <https://skemman.is/handle/1946/37528>, 2020, (Accessed on 10/02/2021).
- [85] Roboflow, “Roboflow application,” <https://app.roboflow.com/datasets>, (Accessed on 04/08/2021).
- [86] Ultralytics, “Github - ultralytics/json2yolo: Convert json annotations into yolo format.” <https://github.com/ultralytics/JSON2YOLO>, (Accessed on 04/08/2021).
- [87] Google, “Preparing your training data | automl vision object detection,” <https://cloud.google.com/vision/automl/object-detection/docs/prepare>, 05 2021, (Accessed on 05/28/2021).
- [88] A. Krizhevsky, “Cifar-10 and cifar-100 datasets,” <https://www.cs.toronto.edu/~kriz/cifar.html>, (Accessed on 05/28/2021).
- [89] Z. Zhang and J. Yu, “Stdgan: Resblock based generative adversarial nets using spectral normalization and two different discriminators,” [https://www.researchgate.net/publication/336714060\\_STDGAN\\_ResBlock\\_Based\\_Generative\\_Adversarial\\_Nets\\_Using\\_Spectral\\_Normalization\\_and\\_Two\\_Different\\_Discriminators/citation/download](https://www.researchgate.net/publication/336714060_STDGAN_ResBlock_Based_Generative_Adversarial_Nets_Using_Spectral_Normalization_and_Two_Different_Discriminators/citation/download), 10 2019, (Accessed on 05/28/2021).
- [90] Ultralytics, “ultralytics/yolov5: Yolov5 in pytorch > onnx > coreml > tflite,” <https://github.com/ultralytics/yolov5>, (Accessed on 05/03/2021).
- [91] —, “Yolov5 tutorial - colab,” [https://colab.research.google.com/github/ultralytics/yolov5/blob/master/tutorial.ipynb#scrollTo=1NcFxRcFdJ\\_O](https://colab.research.google.com/github/ultralytics/yolov5/blob/master/tutorial.ipynb#scrollTo=1NcFxRcFdJ_O), (Accessed on 05/25/2021).
- [92] Z. He, “Github - lynnho/dcgan-lsgan-wgan-gp-dragan-tensorflow-2,” <https://github.com/LynnHo/DCGAN-LSGAN-WGAN-GP-DRAGAN-Tensorflow-2>, (Accessed on 05/24/2021).
- [93] —, “zhenliang he - google scholar,” <https://scholar.google.com/citations?user=fDTTEaAAAAAJ&hl=en>, (Accessed on 05/24/2021).

- [94] —, “Dcgan-lsgan-wgan-gp-dragan-tensorflow-2/license at master · lynnho/dcgan-lsgan-wgan-gp-dragan-tensorflow-2 · github,” <https://github.com/LynnHo/DCGAN-LSGAN-WGAN-GP-DRAGAN-Tensorflow-2/blob/master/LICENSE>, (Accessed on 05/24/2021).
- [95] TensorFlow, “tf.keras.optimizers.adam | tensorflow core v2.5.0,” [https://www.tensorflow.org/api\\_docs/python/tf/keras/optimizers/Adam](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam), (Accessed on 05/26/2021).
- [96] J. Brownlee, “How to control the stability of training neural networks with the batch size,” <https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/>, 08 2020, (Accessed on 05/29/2021).
- [97] M. M. B. X. D. W.-F. S. O. A. C. Ian J. Goodfellow, Jean Pouget-Abadie and Y. Bengio, “[1406.2661] generative adversarial networks,” <https://arxiv.org/abs/1406.2661>, 06 2014, (Accessed on 05/02/2021).
- [98] W. Z. V. C. A. R. Tim Salimans, Ian Goodfellow and X. Chen, “[1606.03498] improved techniques for training gans,” <https://arxiv.org/abs/1606.03498>, 06 2016, (Accessed on 05/02/2021).
- [99] T. U. B. N. Martin Heusel, Hubert Ramsauer and S. Hochreiter, “[1706.08500] gans trained by a two time-scale update rule converge to a local nash equilibrium,” <https://arxiv.org/abs/1706.08500>, 01 2018, (Accessed on 05/02/2021).
- [100] M. M. S. G. Mario Lucic, Karol Kurach and O. Bousquet, “[1711.10337] are gans created equal? a large-scale study,” <https://arxiv.org/abs/1711.10337>, 10 2018, (Accessed on 05/02/2021).
- [101] J. Brownlee, “How to implement the inception score (is) for evaluating gans,” <https://machinelearningmastery.com/how-to-implement-the-inception-score-from-scratch-for-evaluating-generated-images/#:~:text=The%20Inception%20Score%2C%20or%20IS%20for%20short%2C%20is%20an%20objective,by%20generative%20adversarial%20network%20models.&text=They%20developed%20the%20inception%20score,subjective%20human%20evaluation%20of%20images.>, 10 2019, (Accessed on 05/31/2021).
- [102] J. Hui, “Gan — cgan & infogan (using labels to improve gan) | by jonathan hui | medium,” <https://jonathan-hui.medium.com/gan-cgan-infogan-using-labels-to-improve-gan-8ba4de5f9c3d>, 06 2018, (Accessed on 05/31/2021).
- [103] DeepLearning.AI, “Inception score - week 1: Evaluation of gans | coursera,” <https://www.coursera.org/lecture/build-better-generative-adversarial-networks-gans/inception-score-HxtYM>, (Accessed on 05/31/2021).
- [104] D. Mack, “A simple explanation of the inception score | by david mack | octavian | medium,” <https://medium.com/octavian-ai/a-simple-explanation-of-the-inception-score-372dff6a8c7a>, 03 2019, (Accessed on 05/31/2021).
- [105] DeepLearning.AI, “Fréchet inception distance (fid) - week 1: Evaluation of gans | coursera,” <https://www.coursera.org/lecture/build-better-generative-adversarial-networks-gans/frechet-inception-distance-fid-LY8WK>, (Accessed on 05/31/2021).



- [106] J. Brownlee, “How to implement the frechet inception distance (fid) for evaluating gans,” <https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/#:~:text=The%20Frechet%20Inception%20Distance%2C%20or,by%20Martin%20Heusel%2C%20et%20al.,> 10 2019, (Accessed on 05/31/2021).
- [107] A. Borji, “[1802.03446] pros and cons of gan evaluation measures (page 31).” <https://arxiv.org/abs/1802.03446>, 10 2018, (Accessed on 06/01/2021).
- [108] Nvidia, “Teslak80-datasheet.pdf,” <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf>, (Accessed on 05/27/2021).
- [109] J. D. T. D. A. A. E. Deepak Pathak, Philipp Krahenbuhl, “[1604.07379] context encoders: Feature learning by inpainting,” <https://arxiv.org/abs/1604.07379>, 11 2016, (Accessed on 06/05/2021).

# Appendix

Experiment figures: GAN generated objects

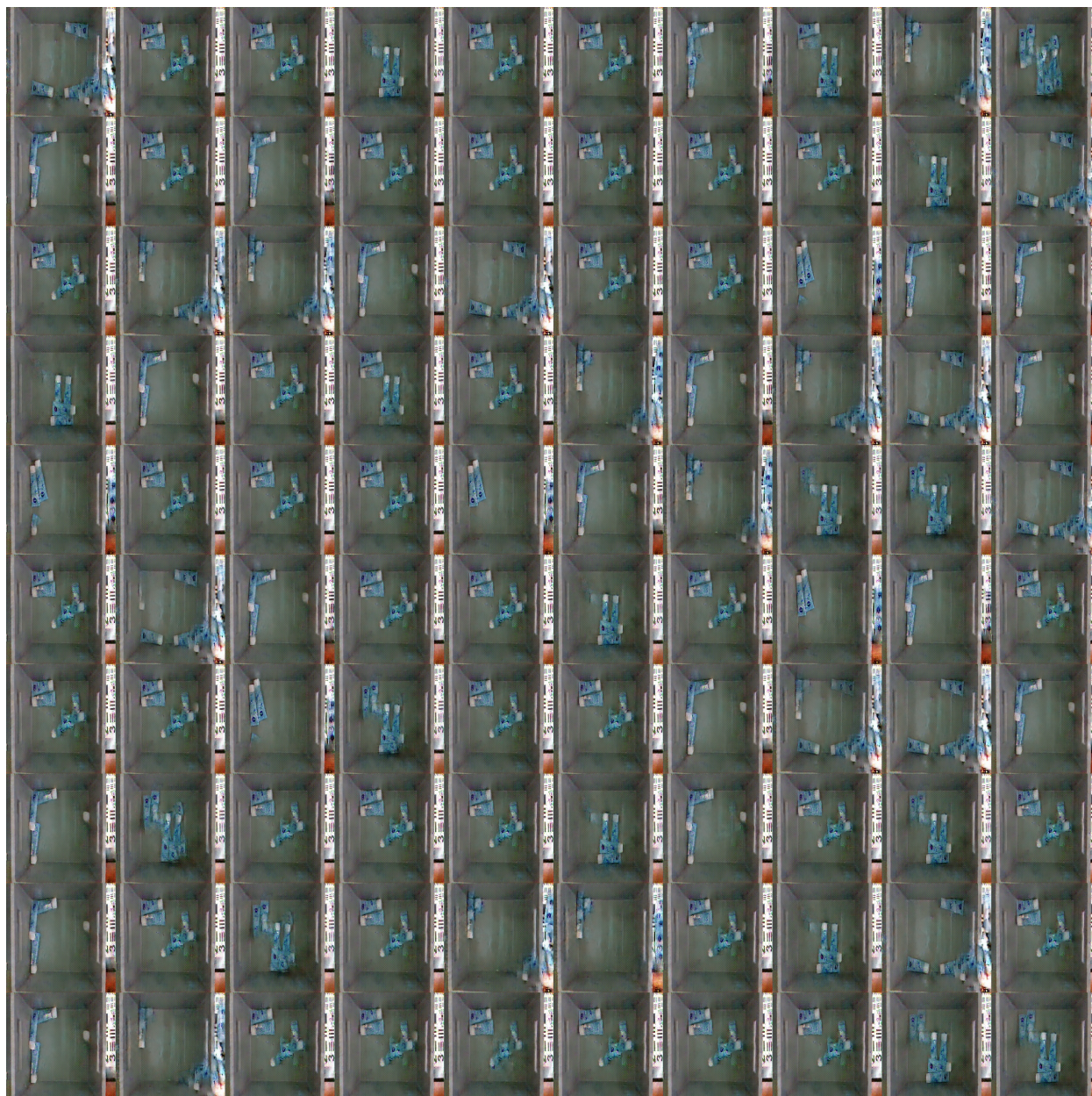


Figure 1: GAN trained with 151 images for 39 500 iterations.

## Experiment figures: WGAN-GP generated objects

- White squares = Too bad quality to be new generations.
- Red squares = Already exist in the training dataset.
- Yellow squares = Represent interesting generations.
- Green squares = New acceptable quality generations

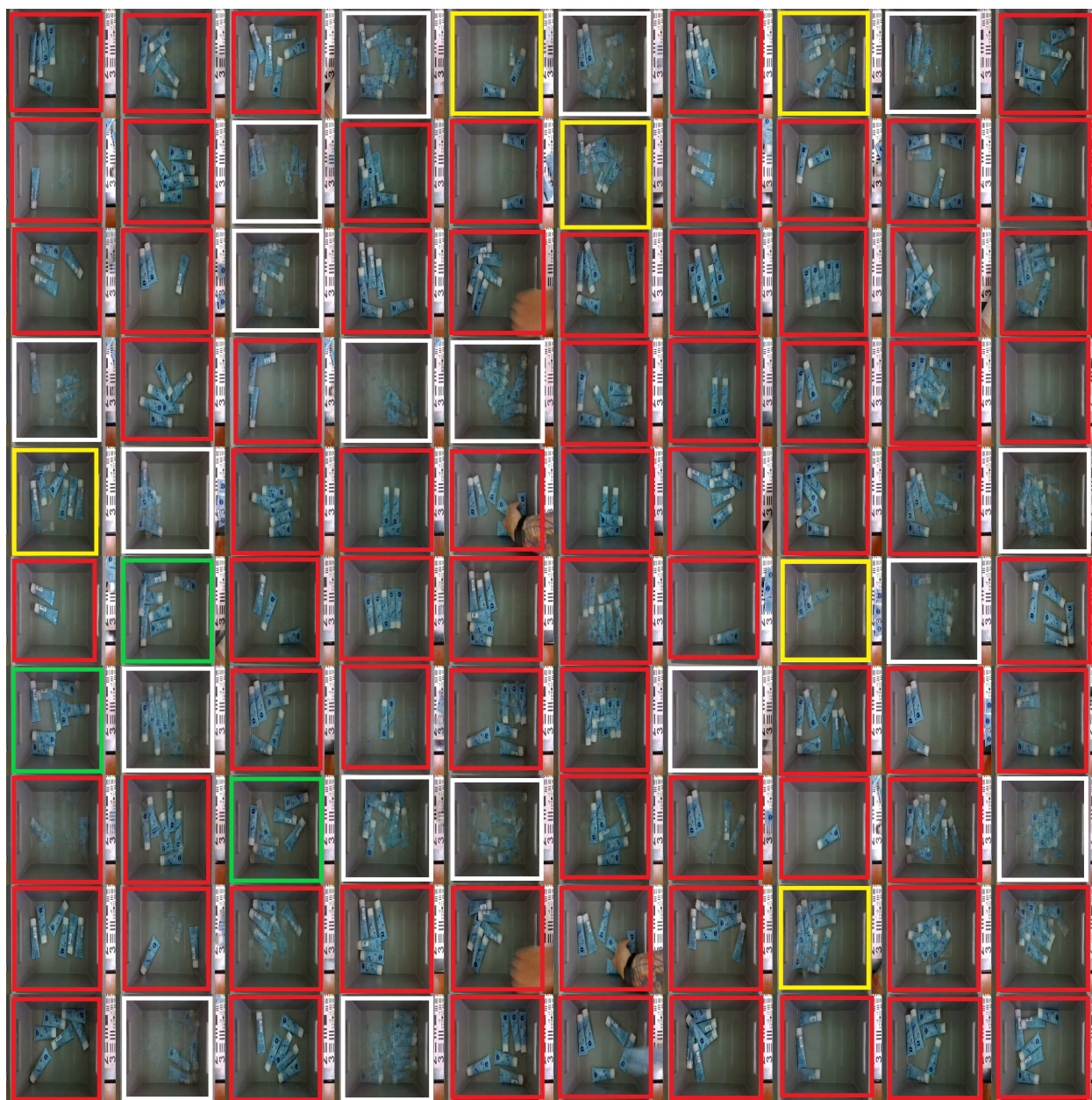


Figure 2: WGAN-GP trained with 151 images for 20 000 iterations.

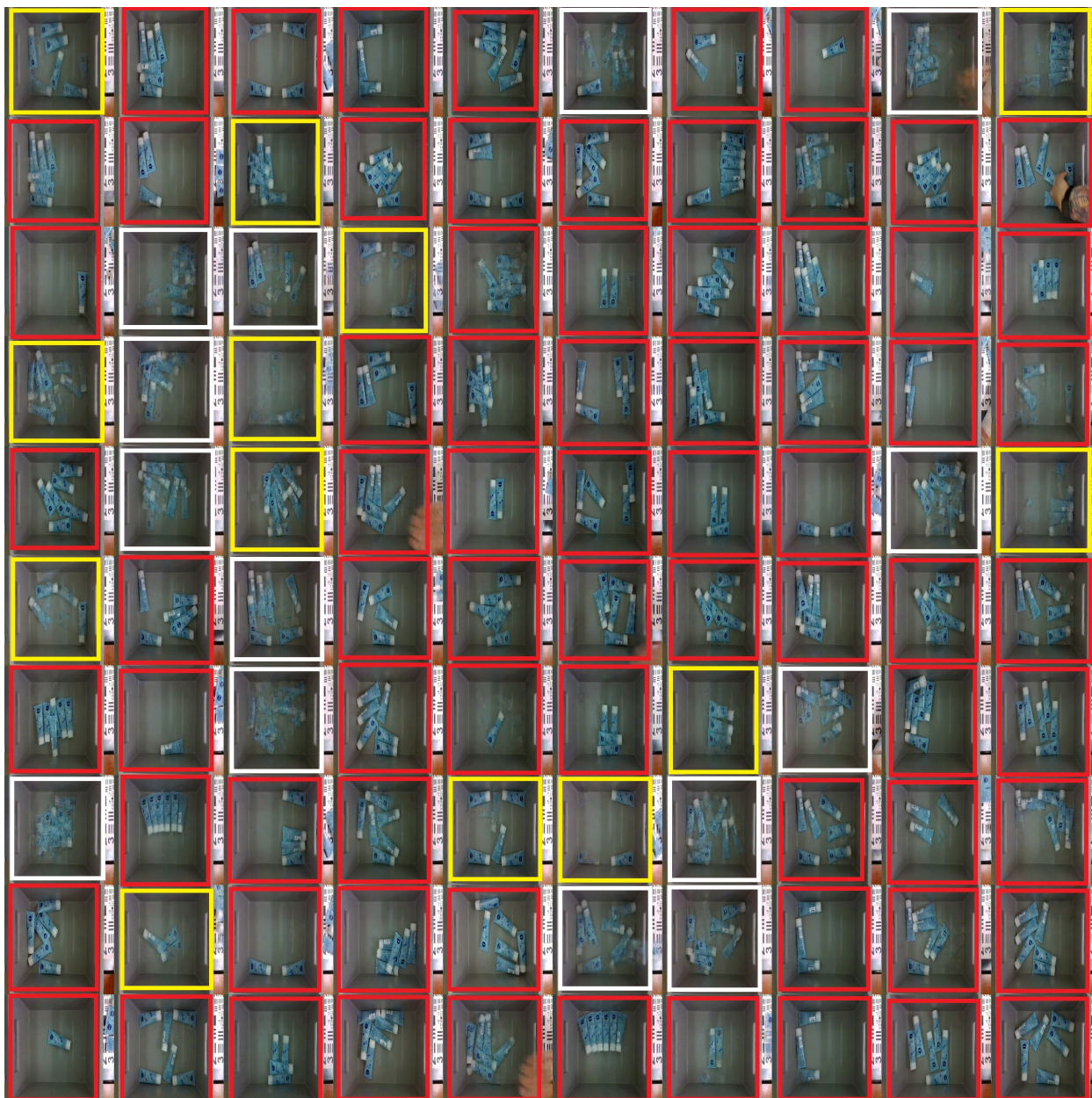


Figure 3: WGAN-GP trained with 151 images for 30 000 iterations.

# Experiment figures: WGAN-GP with 10 images

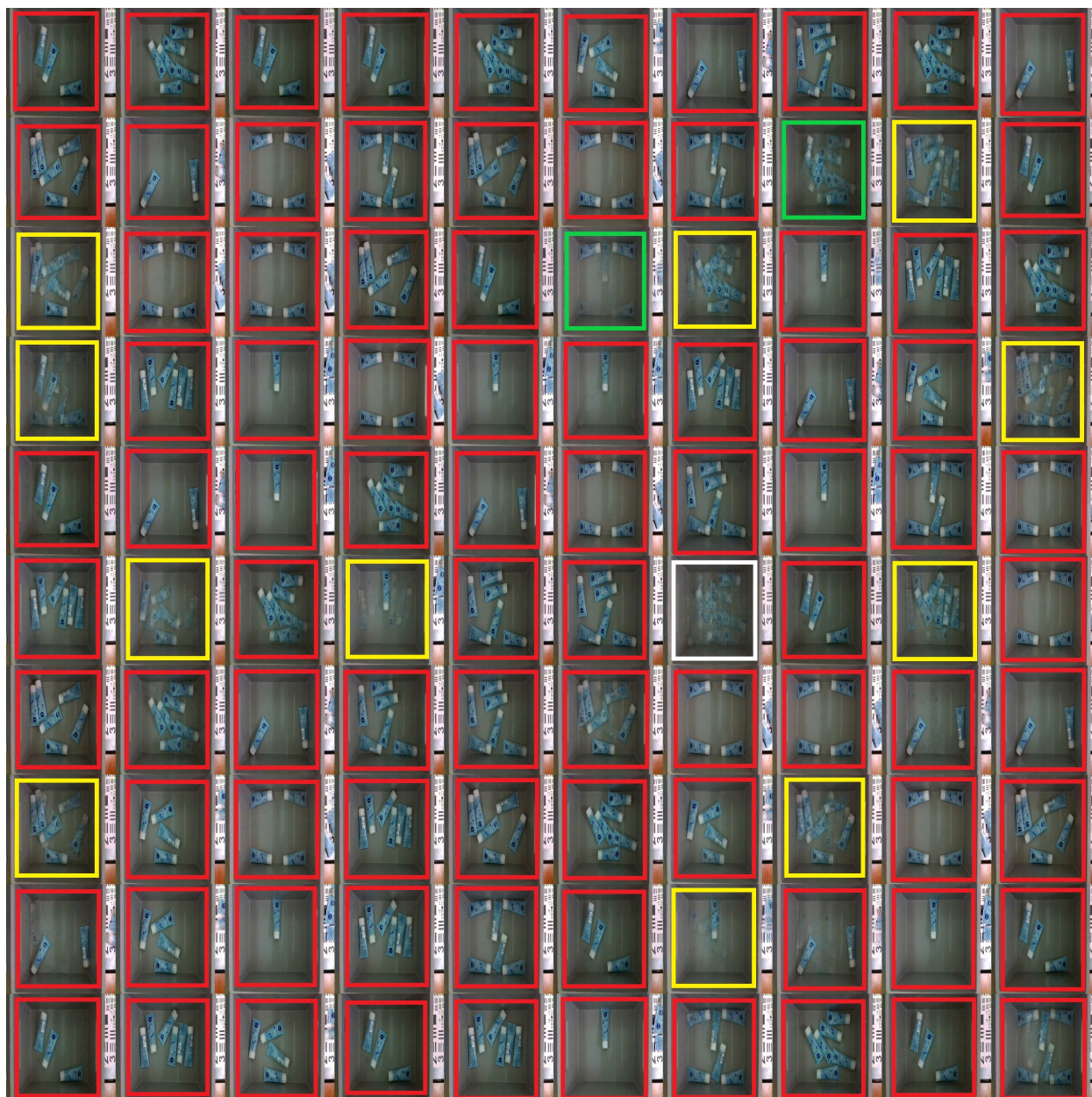


Figure 4: 100 generated samples after 8 000 iterations of training on 10 images

# Experiment figures: WGAN-GP with 10 images

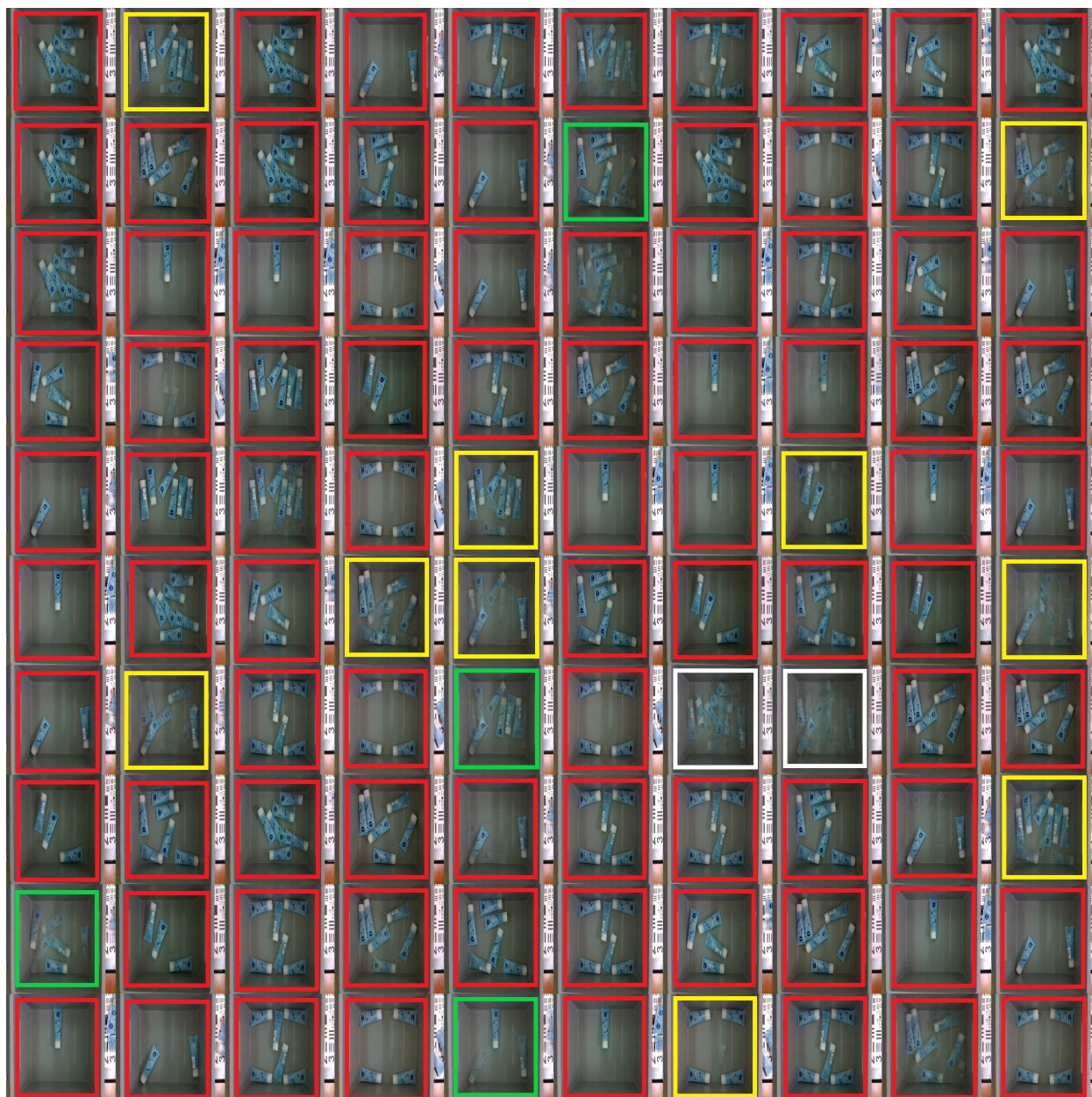


Figure 5: 100 generated samples after 9 000 iterations of training on 10 images

Experiment figures: WGAN-GP with 10 images

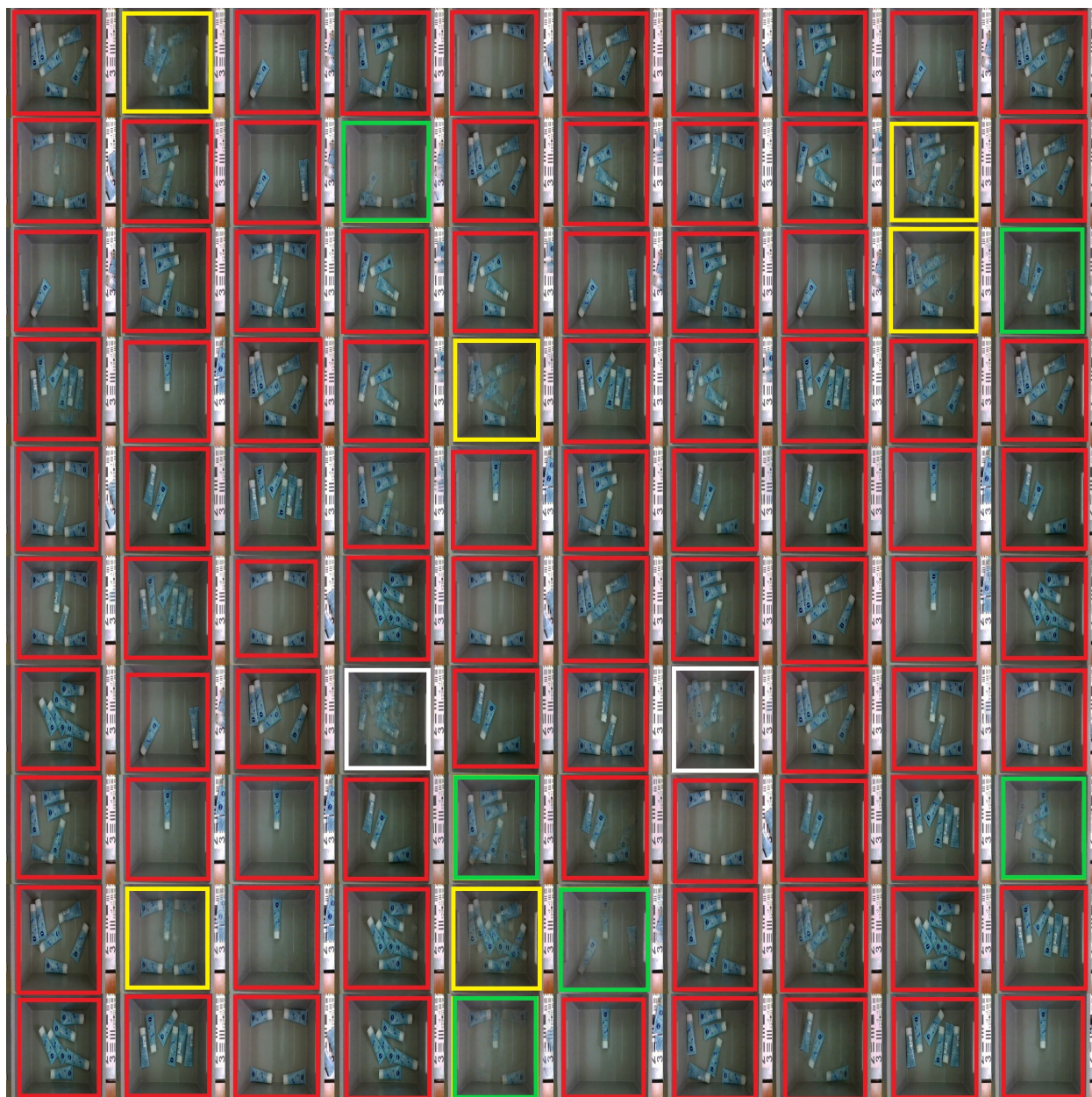


Figure 6: 100 generated samples after 10 000 iterations of training on 10 images

Experiment figures: Pre-trained WGAN-GP with 10 images

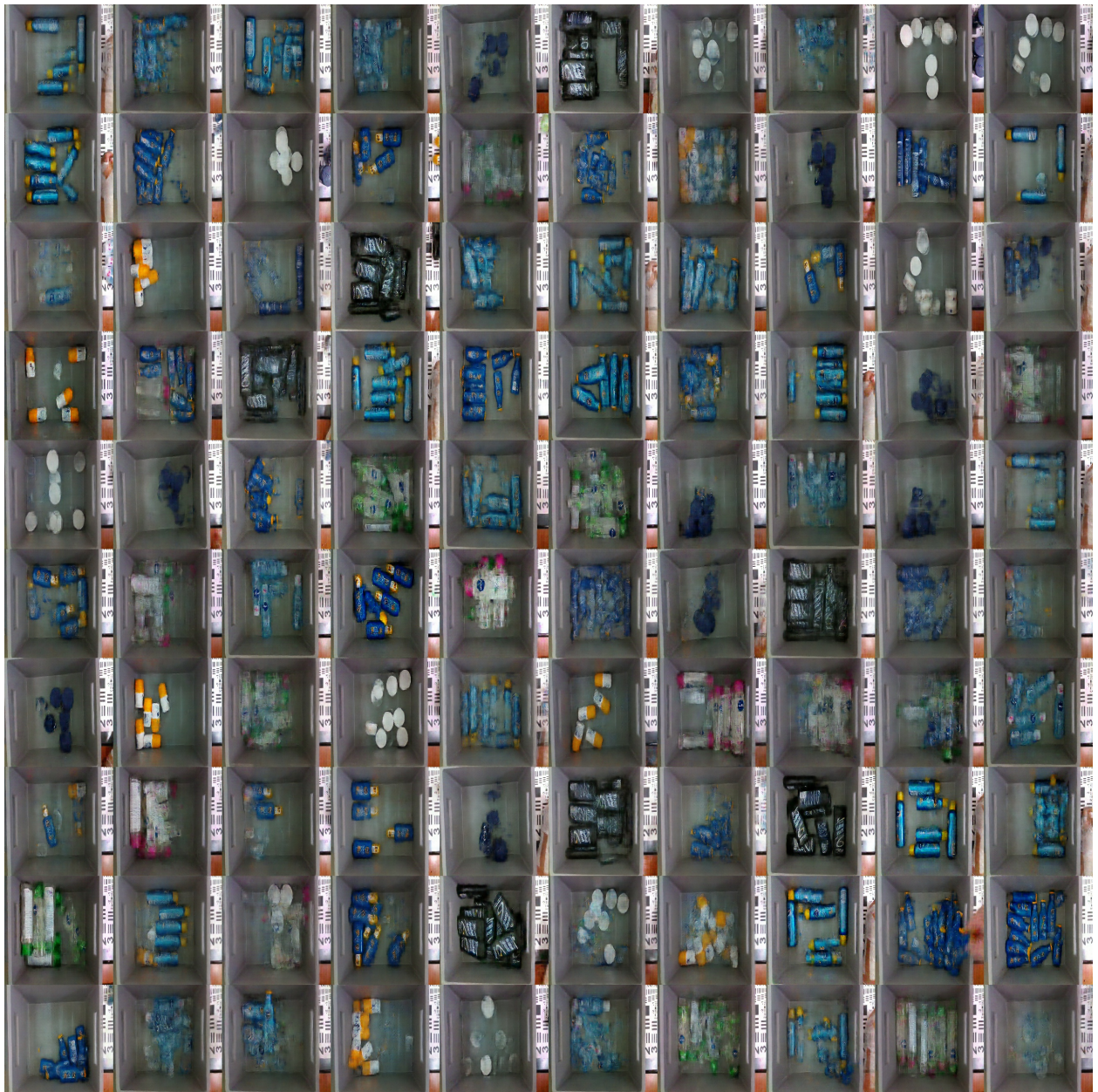


Figure 7: Pre-trained model after 675 000 iterations of training on 1 801 images



## Experiment figures: Pre-trained WGAN-GP with 10 images

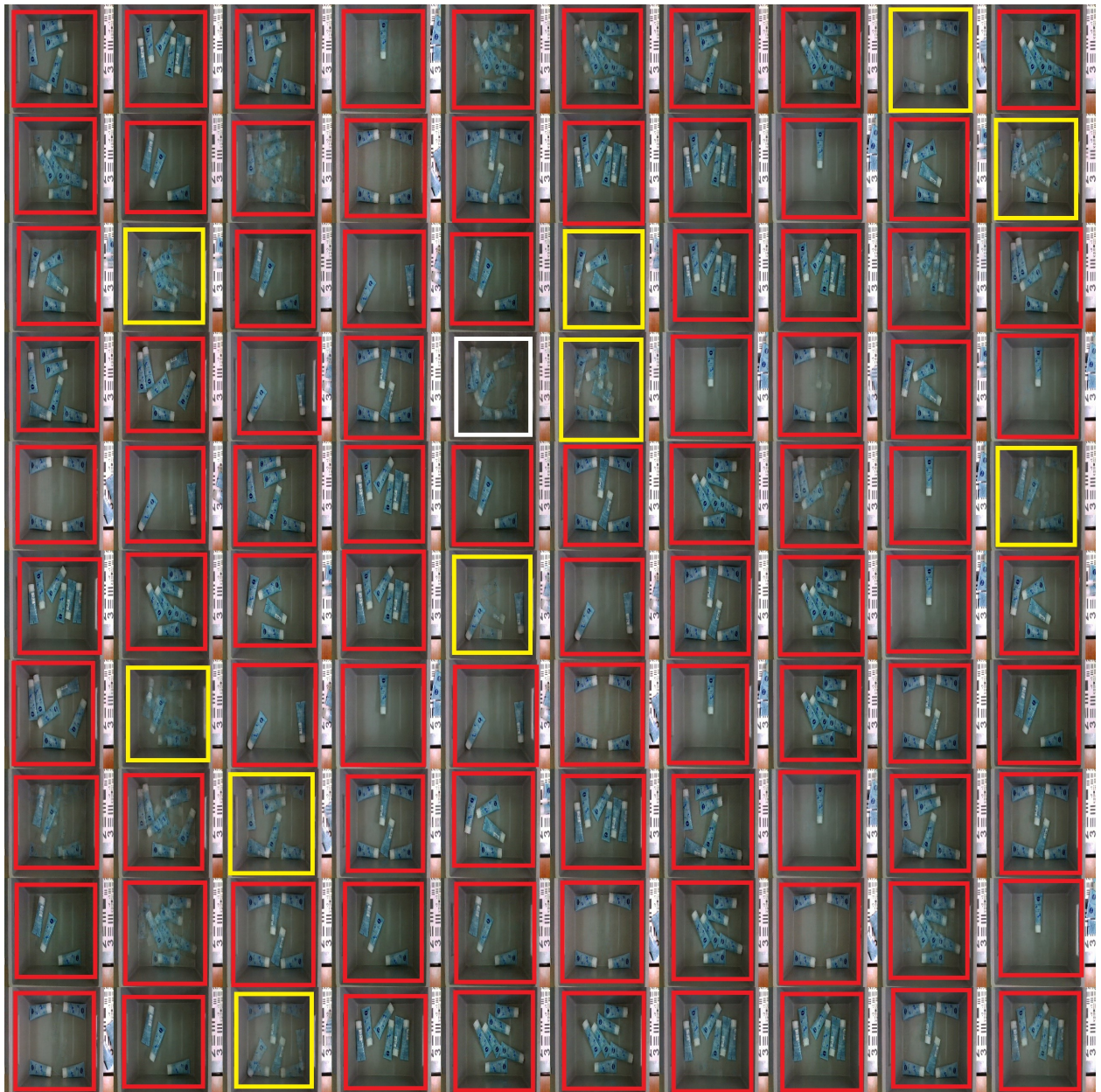


Figure 8: 100 generated samples after 8 000 (683 000) iterations of continued training with a pre-trained model on 10 images

## Experiment figures: Pre-trained WGAN-GP with 10 images

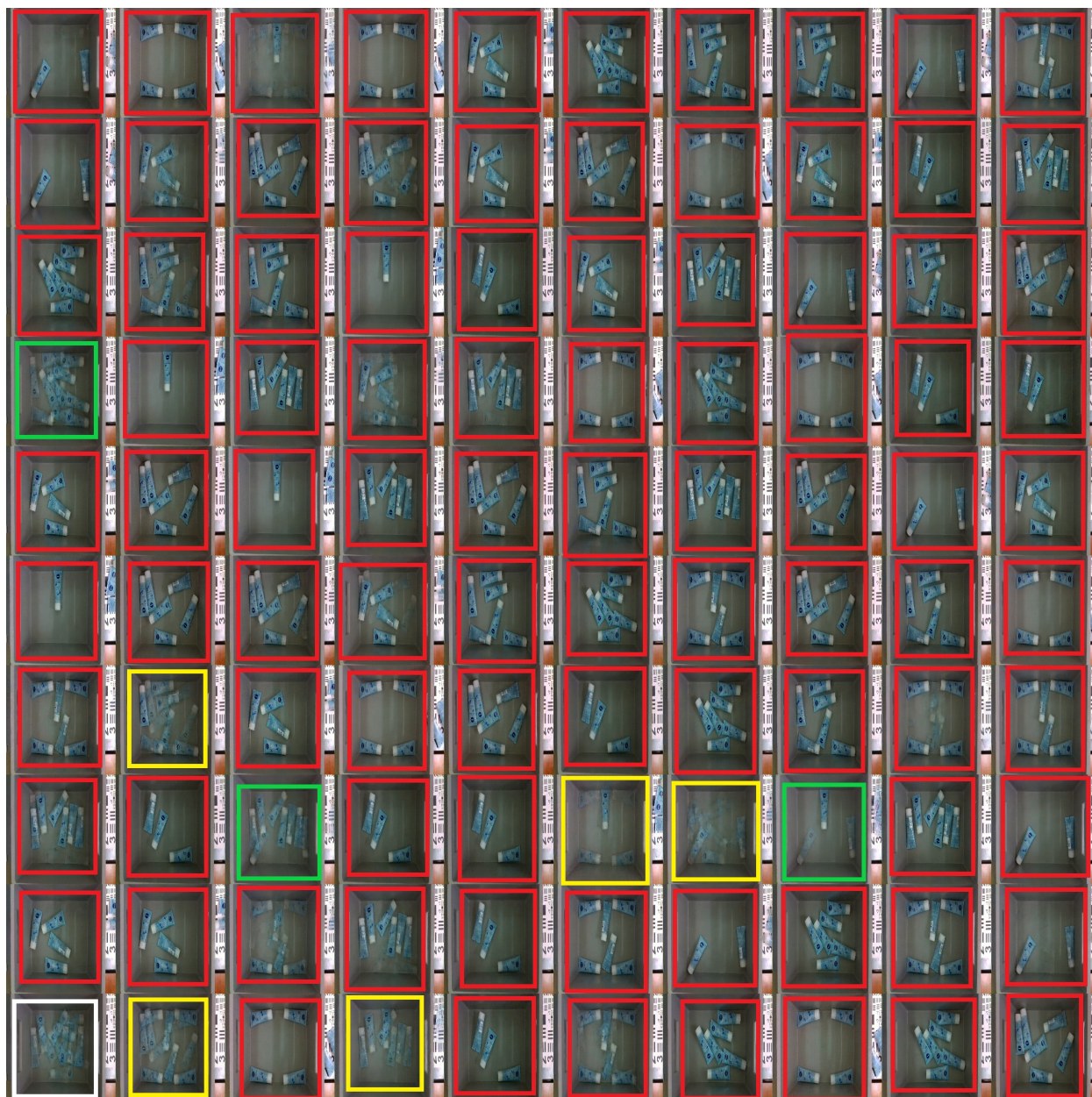


Figure 9: 100 generated samples after 9 000 (684 000) iterations of continued training with a pre-trained model on 10 images

## Experiment figures: Pre-trained WGAN-GP with 10 images

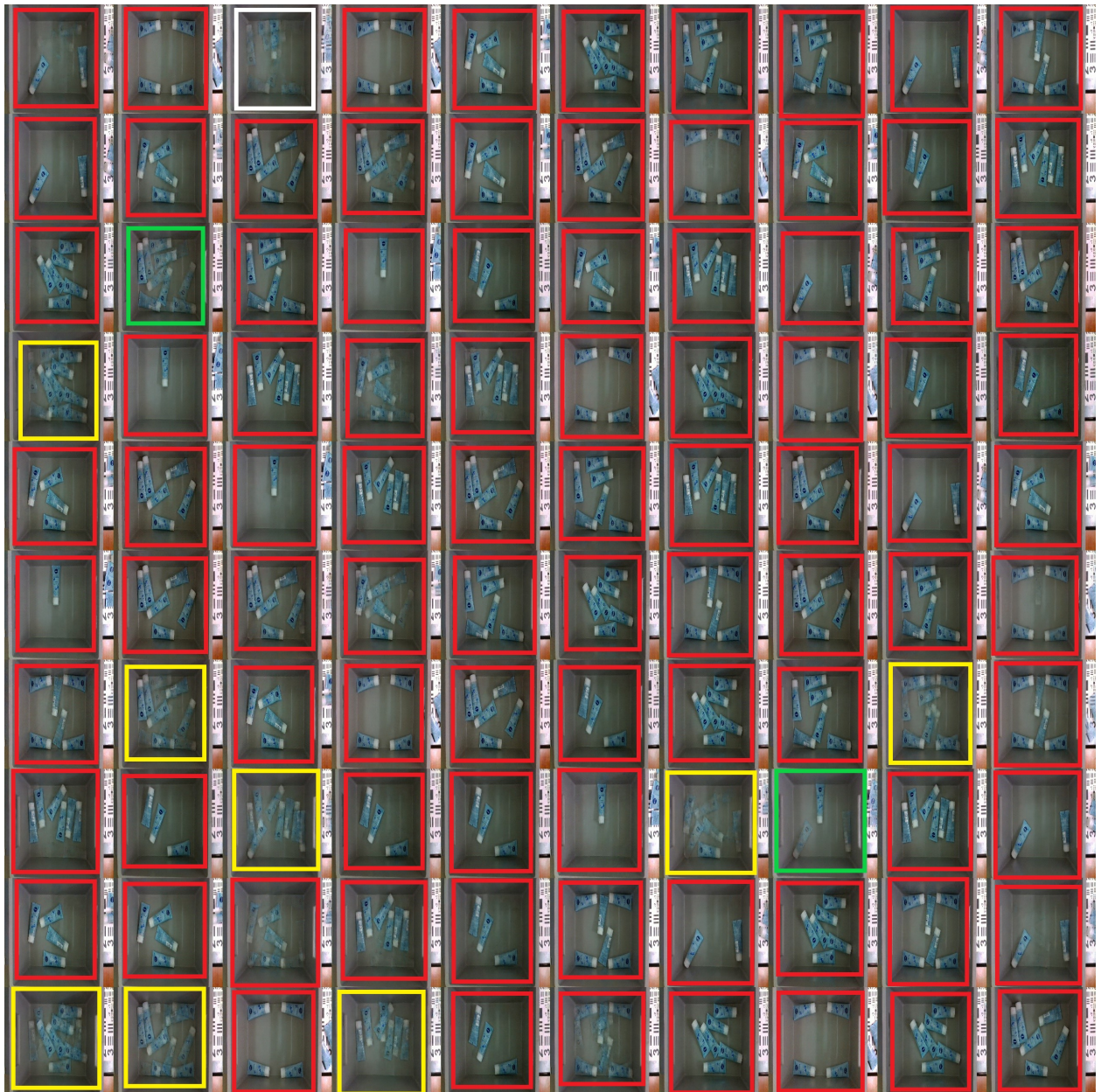


Figure 10: 100 generated samples after 10 000 (685 000) iterations of continued training with a pre-trained model on 10 images

## Experiment figures: Training with generated images in the dataset

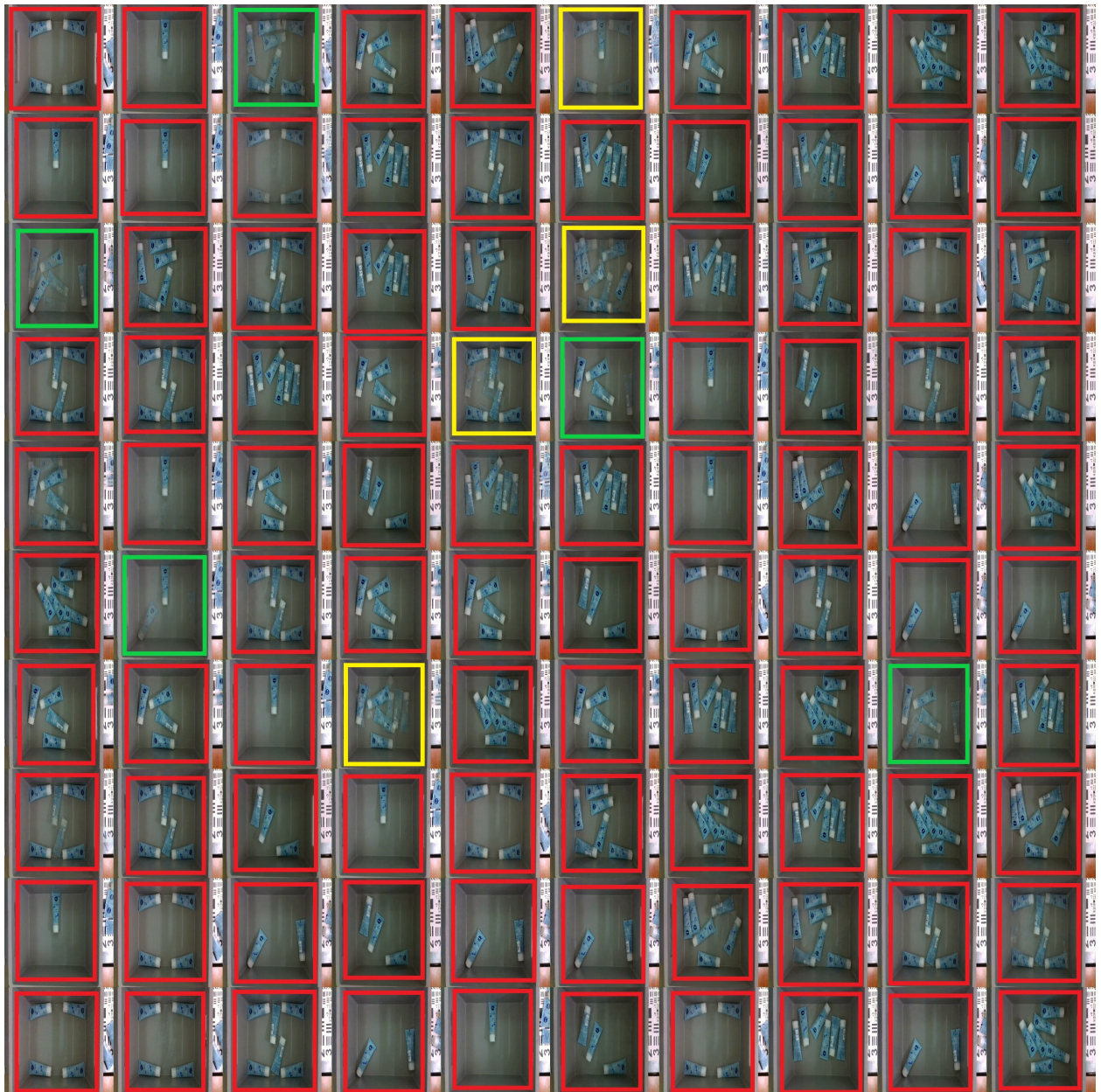


Figure 11: 100 generated samples after 8 000 (693 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images

## Experiment figures: Training with generated images in the dataset

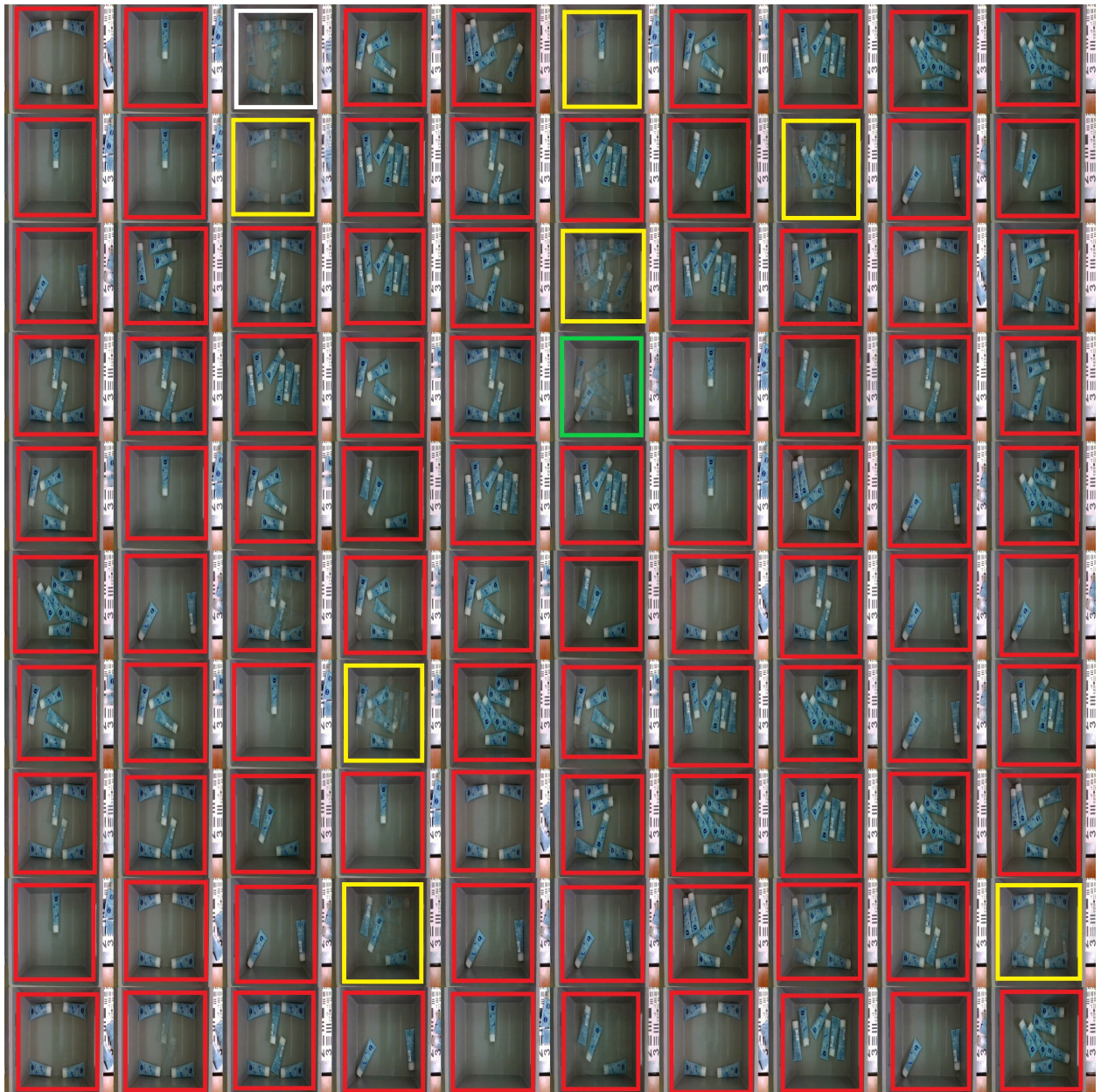


Figure 12: 100 generated samples after 9 000 (694 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images

## Experiment figures: Training with generated images in the dataset



Figure 13: 100 generated samples after 10 000 (695 000) more iterations of continued training with a pre-trained model on 10 real and 10 fake images

Experiment figures: Expanding dataset with 10 images

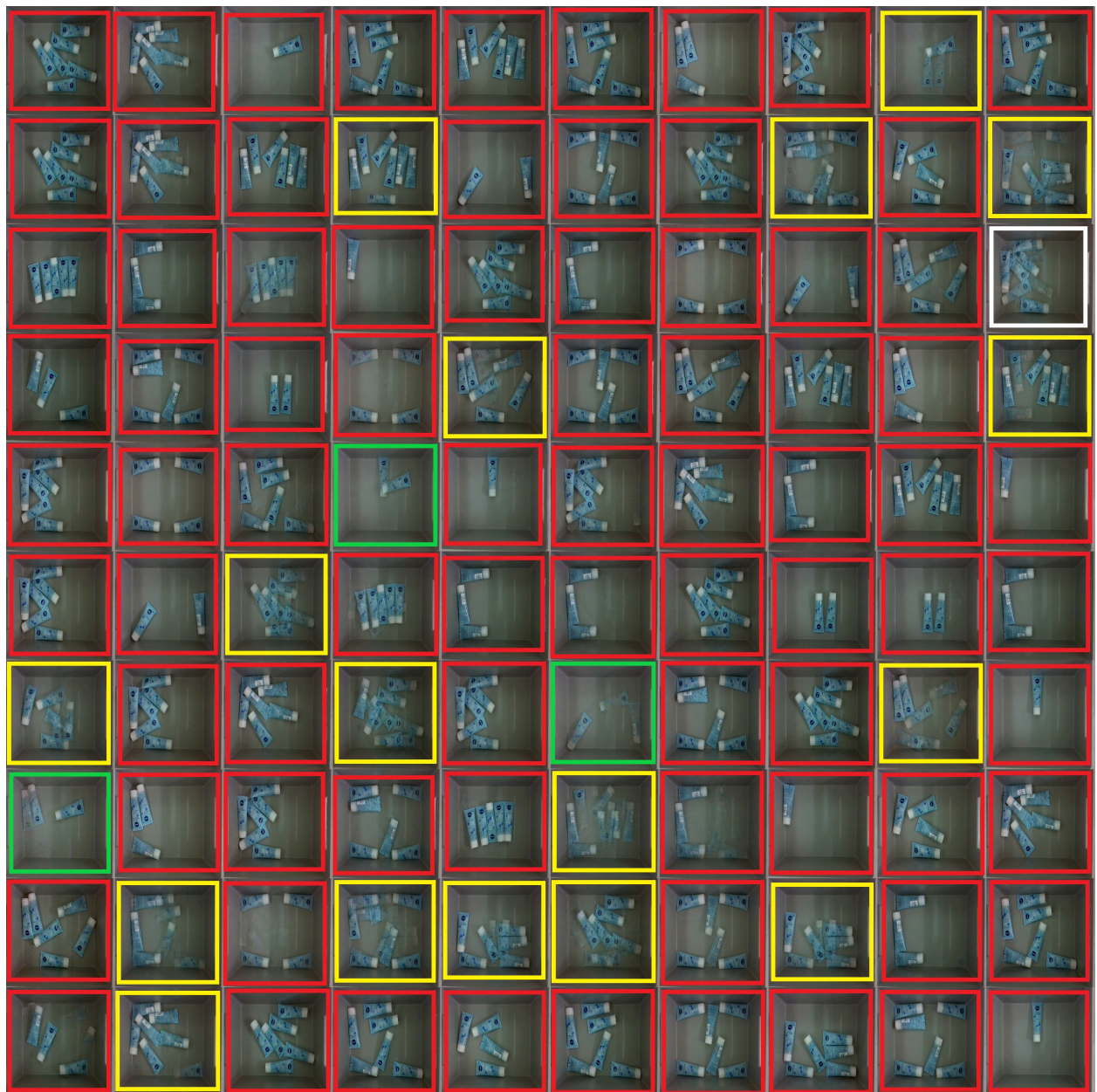


Figure 14: 100 generated samples after 18 000 iterations of training with 20 real images

Experiment figures: Expanding dataset with 10 images

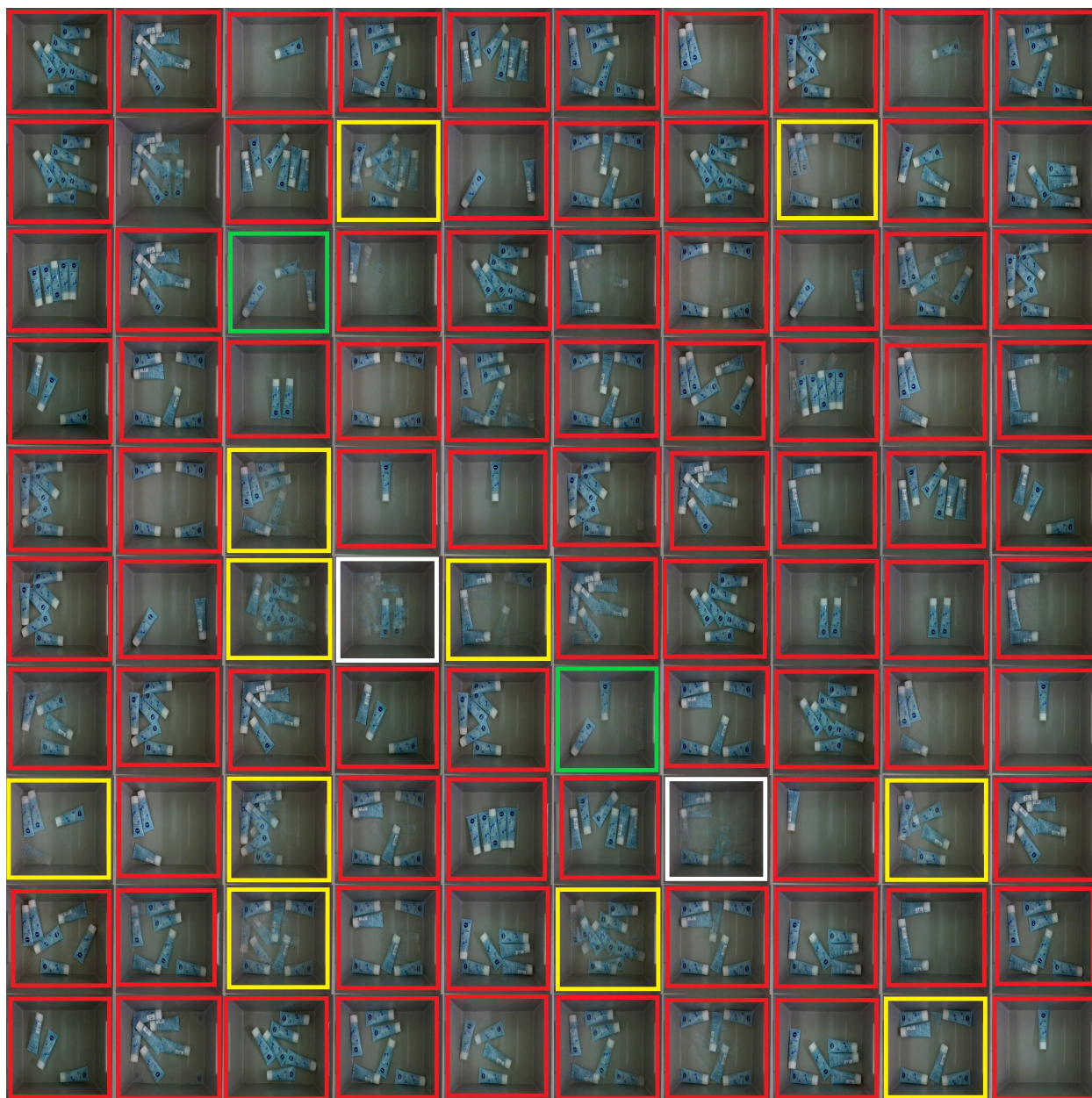


Figure 15: 100 generated samples after 19 000 iterations of training with 20 real images



## Experiment figures: Expanding dataset with 10 images

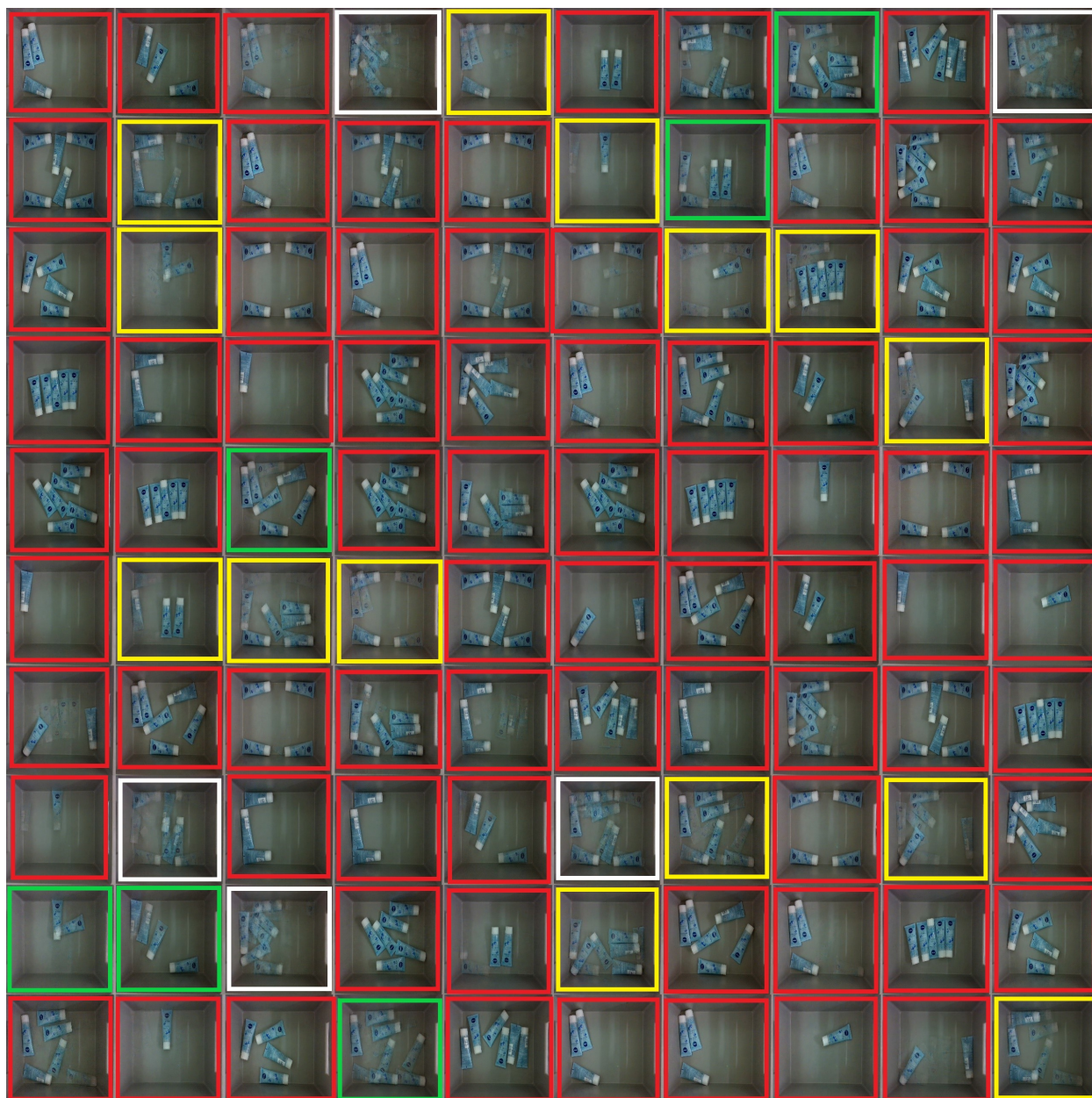


Figure 16: 100 generated samples after 20 000 iterations of training with 20 real images

### Code used for this thesis:

"main\_training" is the code for GAN training, the required setup is explained in chapter 3.2. Additional, the "YOLOv5\_JEB\_v2\_2" is the code for object detector.

The code for this jupyter notebook are built from python scrips provided in the GitHub repository: <https://github.com/LynnHo/DCGAN-LSGAN-WGAN-GP-DRAGAN-Tensorflow-2>

Import's:

```
In [25]: import functools
import tensorflow as tf
import tensorflow.keras as keras
import tqdm
import datetime
import fnmatch
import os
import glob as _glob
import sys
import multiprocessing

import tensorflow_addons as tfa
import tensorflow.keras as keras

import numpy as np
import skimage.color as color
import skimage.transform as transform

import skimage.io as iio
```

Tensorflow GPU:

```
In [26]: print(f"Tensor Flow Version: {tf.__version__}")
gpu = len(tf.config.list_physical_devices('GPU'))>0
print("GPU is", "available" if gpu else "NOT AVAILABLE")
```

Tensor Flow Version: 2.4.1  
GPU is available

Parameters:

```
In [45]: BATCH_SIZE = 8 # Batch size
IMG_RESIZE = 256 # Size of images (square)
IMG_PATHS = '' # Image path to custom dataset
IMG_FORMAT = '*.png' #Image format in dataset
EPOCHS = int() # Number of training epochs
LR = float(0.0002) # Learning rate
BETA_1 = float(0.5) # Beta_1
N_D = int() # N_D, ratio between discriminator and generator
Z_DIM = int(128)
ADVERSARIAL_LOSS_MODE = 'wgan' #'gan', 'hinge_v1', 'hinge_v2', 'lsgan', 'wgan' # GAN version
GRADIENT_PENALTY_MODE = 'wgan-gp' #'none', 'dragan', 'wgan-gp' # Gradient penalty mode
GRADIENT_PENALTY_WEIGHT = float(10.0) # Gradient penalty weight
PRINT_SAMPLE = 250
EXPERIMENT_NAME = '' # Name for experiment
DATASET_NAME = '' # Name for dataset
```

Input data:

```
In [28]: def glob(dir, pats, recursive=False): # faster than match, python3 only
    pats = pats if isinstance(pats, (list, tuple)) else [pats]
    matches = []
    for pat in pats:
        matches += _glob.glob(os.path.join(dir, pat), recursive=recursive)
    return matches

def batch_dataset(dataset, batch_size, drop_remainder=True, n_prefetch_batch=1, filter_fn=None, map_fn=None,
                 n_map_threads=None, filter_after_map=False, shuffle=True, shuffle_buffer_size=None, repeat=None):
    # set defaults
    if n_map_threads is None:
        n_map_threads = multiprocessing.cpu_count()
    if shuffle and shuffle_buffer_size is None:
        shuffle_buffer_size = max(batch_size * 128, 2048) # set the minimum buffer size as 2048

    # [*] it is efficient to conduct `shuffle` before `map`/`filter` because `map`/`filter` is sometimes costly
    if shuffle:
        dataset = dataset.shuffle(shuffle_buffer_size)

    if not filter_after_map:
        if filter_fn:
            dataset = dataset.filter(filter_fn)

        if map_fn:
            dataset = dataset.map(map_fn, num_parallel_calls=n_map_threads)

    else: # [*] this is slower
        if map_fn:
```

```

        dataset = dataset.map(map_fn, num_parallel_calls=n_map_threads)

    if filter_fn:
        dataset = dataset.filter(filter_fn)

dataset = dataset.batch(batch_size, drop_remainder=drop_remainder)

dataset = dataset.repeat(repeat).prefetch(n_prefetch_batch)

return dataset

def memory_data_batch_dataset(memory_data, batch_size, drop_remainder=True, n_prefetch_batch=1, filter_fn=None,
                              map_fn=None, n_map_threads=None, filter_after_map=False, shuffle=True,
                              shuffle_buffer_size=None, repeat=None):
    """Batch dataset of memory data.

    Parameters
    -----
    memory_data : nested structure of tensors/ndarrays/lists

    """
    dataset = tf.data.Dataset.from_tensor_slices(memory_data)
    dataset = batch_dataset(dataset,
                            batch_size,
                            drop_remainder=drop_remainder,
                            n_prefetch_batch=n_prefetch_batch,
                            filter_fn=filter_fn,
                            map_fn=map_fn,
                            n_map_threads=n_map_threads,
                            filter_after_map=filter_after_map,
                            shuffle=shuffle,
                            shuffle_buffer_size=shuffle_buffer_size,
                            repeat=repeat)

    return dataset

def disk_image_batch_dataset(img_paths, batch_size, labels=None, drop_remainder=True, n_prefetch_batch=1,
                              filter_fn=None, map_fn=None, n_map_threads=None, filter_after_map=False, shuffle=True,
                              shuffle_buffer_size=None, repeat=None):
    """Batch dataset of disk image for PNG and JPEG.

    Parameters
    -----
    img_paths : 1d-tensor/ndarray/list of str
    labels : nested structure of tensors/ndarrays/lists

    """
    if labels is None:
        memory_data = img_paths
    else:
        memory_data = (img_paths, labels)

    def parse_fn(path, *label):
        img = tf.io.read_file(path)
        img = tf.image.decode_png(img, 3) # fix channels to 3
        return (img,) + label

    if map_fn: # fuse `map_fn` and `parse_fn`
        def map_fn_(*args):
            return map_fn(*parse_fn(*args))
    else:
        map_fn_ = parse_fn

    dataset = memory_data_batch_dataset(memory_data,
                                       batch_size,
                                       drop_remainder=drop_remainder,
                                       n_prefetch_batch=n_prefetch_batch,
                                       filter_fn=filter_fn,
                                       map_fn=map_fn_,
                                       n_map_threads=n_map_threads,
                                       filter_after_map=filter_after_map,
                                       shuffle=shuffle,
                                       shuffle_buffer_size=shuffle_buffer_size,
                                       repeat=repeat)

    return dataset

def make_custom_datset(img_paths, batch_size, resize=64, drop_remainder=True, shuffle=True, repeat=1):
    @tf.function
    def _map_fn(img):
        # =====
        # =                  custom          =
        # =====
        #img = ... # custom preprocessings, should output img in [0.0, 255.0]
        # =====
        # =                  custom          =
        # =====
        img = tf.image.resize(img, [resize, resize])

```

```

img = tf.clip_by_value(img, 0, 255)
img = img / 127.5 - 1
return img

dataset = disk_image_batch_dataset(img_paths,
                                   batch_size,
                                   drop_remainder=drop_remainder,
                                   map_fn=_map_fn,
                                   shuffle=shuffle,
                                   repeat=repeat)

img_shape = (resize, resize, 3)
len_dataset = len(img_paths) // batch_size

return dataset, img_shape, len_dataset

```

```

In [29]: img_paths = glob(IMG_PATHS, IMG_FORMAT)
dataset, shape, len_dataset = make_custom_datset(img_paths, BATCH_SIZE, resize=IMG_RESIZE) # make_custom_datset(img_p
n_G_upsamplings = n_D_downsamplings = 6 #... # 3 for 32x32 and 4 for 64x64

```

Output data:

```

In [30]: def mkdir(paths):
if not isinstance(paths, (list, tuple)):
paths = [paths]
for path in paths:
if not os.path.exists(path):
os.makedirs(path)

```

```

In [31]: if EXPERIMENT_NAME == 'none':
EXPERIMENT_NAME = '%s_%s' % (DATASET_NAME, ADVERSERIAL_LOSS_MODE)
if GRADIENT_PENALTY_MODE != 'none':
EXPERIMENT_NAME += '%s' % GRADIENT_PENALTY_MODE
output_dir = os.path.join('output', EXPERIMENT_NAME)
mkdir(output_dir)

# save settings
#py.args_to_yaml(py.join(output_dir, 'settings.yaml'), args) #!!!!!!!!!!!!!!!!!!!!!!

```

```

In [32]: # setup the normalization function for discriminator
if GRADIENT_PENALTY_MODE == 'none':
D_NORM = 'batch_norm'
elif GRADIENT_PENALTY_MODE in ['dragan', 'wgan-gp']: # cannot use batch normalization with gradient penalty
D_NORM = 'layer_norm'

```

```

In [33]: def _get_norm_layer(norm):
if norm == 'none':
return lambda: lambda x: x
elif norm == 'batch_norm':
return keras.layers.BatchNormalization
elif norm == 'instance_norm':
return tfa.layers.InstanceNormalization
elif norm == 'layer_norm':
return keras.layers.LayerNormalization

def ConvGenerator(input_shape=(1, 1, 128),
                 output_channels=3,
                 dim=64,
                 n_upsamplings=4,
                 norm='batch_norm',
                 name='ConvGenerator'):
Norm = _get_norm_layer(norm)

# 0
h = inputs = keras.Input(shape=input_shape)

# 1: 1x1 -> 4x4
d = min(dim * 2 ** (n_upsamplings - 1), dim * 8)
h = keras.layers.Conv2DTranspose(d, 4, strides=1, padding='valid', use_bias=False)(h)
h = Norm()(h)
h = tf.nn.relu(h) # or h = keras.Layers.ReLU()(h)

# 2: upsamplings, 4x4 -> 8x8 -> 16x16 -> ...
for i in range(n_upsamplings - 1):
d = min(dim * 2 ** (n_upsamplings - 2 - i), dim * 8)
h = keras.layers.Conv2DTranspose(d, 4, strides=2, padding='same', use_bias=False)(h)
h = Norm()(h)
h = tf.nn.relu(h) # or h = keras.Layers.ReLU()(h)

h = keras.layers.Conv2DTranspose(output_channels, 4, strides=2, padding='same')(h)
h = tf.tanh(h) # or h = keras.Layers.Activation('tanh')(h)

```

```

return keras.Model(inputs=inputs, outputs=h, name=name)

def ConvDiscriminator(input_shape=(64, 64, 3),
                     dim=64,
                     n_downsamplings=4,
                     norm='batch_norm',
                     name='ConvDiscriminator'):
    Norm = _get_norm_layer(norm)

    # 0
    h = inputs = keras.Input(shape=input_shape)

    # 1: downsamplings, ... -> 16x16 -> 8x8 -> 4x4
    h = keras.layers.Conv2D(dim, 4, strides=2, padding='same')(h)
    h = tf.nn.leaky_relu(h, alpha=0.2) # or keras.Layers.LeakyReLU(alpha=0.2)(h)

    for i in range(n_downsamplings - 1):
        d = min(dim * 2 ** (i + 1), dim * 8)
        D = keras.layers.Conv2D(d, 4, strides=2, padding='same', use_bias=False)(h)
        h = Norm()(h)
        h = tf.nn.leaky_relu(h, alpha=0.2) # or h = keras.Layers.LeakyReLU(alpha=0.2)(h)

    # 2: Logit
    print(h)
    h = keras.layers.Conv2D(1, 4, strides=1, padding='valid')(h)

    return keras.Model(inputs=inputs, outputs=h, name=name)

```

In [34]:

```

# networks
G = ConvGenerator(input_shape=(1, 1, Z_DIM), output_channels=shape[-1], n_upsamplings=n_G_upsamplings, name='G%s' % DATASET_NAME)
D = ConvDiscriminator(input_shape=shape, n_downsamplings=n_D_downsamplings, norm=D_NORM, name='D%s' % DATASET_NAME)
G.summary()
D.summary()

```

KerasTensor(type\_spec=TensorSpec(shape=(None, 4, 4, 512), dtype=tf.float32, name=None), name='tf.nn.leaky\_relu\_11/LeakyRelu:0', description="created by layer 'tf.nn.leaky\_relu\_11'")  
Model: "G\_Resize\_256\_all\_class\_except\_1\_4\_mix"

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 1, 1, 128)]	0
conv2d_transpose_7 (Conv2DTr	(None, 4, 4, 512)	1048576
batch_normalization_6 (Batch	(None, 4, 4, 512)	2048
tf.nn.relu_6 (TFOpLambda)	(None, 4, 4, 512)	0
conv2d_transpose_8 (Conv2DTr	(None, 8, 8, 512)	4194304
batch_normalization_7 (Batch	(None, 8, 8, 512)	2048
tf.nn.relu_7 (TFOpLambda)	(None, 8, 8, 512)	0
conv2d_transpose_9 (Conv2DTr	(None, 16, 16, 512)	4194304
batch_normalization_8 (Batch	(None, 16, 16, 512)	2048
tf.nn.relu_8 (TFOpLambda)	(None, 16, 16, 512)	0
conv2d_transpose_10 (Conv2DT	(None, 32, 32, 256)	2097152
batch_normalization_9 (Batch	(None, 32, 32, 256)	1024
tf.nn.relu_9 (TFOpLambda)	(None, 32, 32, 256)	0
conv2d_transpose_11 (Conv2DT	(None, 64, 64, 128)	524288
batch_normalization_10 (Batc	(None, 64, 64, 128)	512
tf.nn.relu_10 (TFOpLambda)	(None, 64, 64, 128)	0
conv2d_transpose_12 (Conv2DT	(None, 128, 128, 64)	131072
batch_normalization_11 (Batc	(None, 128, 128, 64)	256
tf.nn.relu_11 (TFOpLambda)	(None, 128, 128, 64)	0
conv2d_transpose_13 (Conv2DT	(None, 256, 256, 3)	3075
tf.math.tanh_1 (TFOpLambda)	(None, 256, 256, 3)	0
=====		
Total params: 12,200,707		
Trainable params: 12,196,739		
Non-trainable params: 3,968		

Model: "D\_Resized\_256\_all\_class\_except\_1\_4\_mix"

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 256, 256, 3)]	0
conv2d_7 (Conv2D)	(None, 128, 128, 64)	3136
tf.nn.leaky_relu_6 (TFOpLamb)	(None, 128, 128, 64)	0
conv2d_8 (Conv2D)	(None, 64, 64, 128)	131072
layer_normalization_5 (Layer)	(None, 64, 64, 128)	256
tf.nn.leaky_relu_7 (TFOpLamb)	(None, 64, 64, 128)	0
conv2d_9 (Conv2D)	(None, 32, 32, 256)	524288
layer_normalization_6 (Layer)	(None, 32, 32, 256)	512
tf.nn.leaky_relu_8 (TFOpLamb)	(None, 32, 32, 256)	0
conv2d_10 (Conv2D)	(None, 16, 16, 512)	2097152
layer_normalization_7 (Layer)	(None, 16, 16, 512)	1024
tf.nn.leaky_relu_9 (TFOpLamb)	(None, 16, 16, 512)	0
conv2d_11 (Conv2D)	(None, 8, 8, 512)	4194304
layer_normalization_8 (Layer)	(None, 8, 8, 512)	1024
tf.nn.leaky_relu_10 (TFOpLam)	(None, 8, 8, 512)	0
conv2d_12 (Conv2D)	(None, 4, 4, 512)	4194304
layer_normalization_9 (Layer)	(None, 4, 4, 512)	1024
tf.nn.leaky_relu_11 (TFOpLam)	(None, 4, 4, 512)	0
conv2d_13 (Conv2D)	(None, 1, 1, 1)	8193

=====  
 Total params: 11,156,289  
 Trainable params: 11,156,289  
 Non-trainable params: 0  
 =====

Loss functions

In [35]:

```
def get_gan_losses_fn():
    bce = tf.losses.BinaryCrossentropy(from_logits=True)

    def d_loss_fn(r_logit, f_logit):
        r_loss = bce(tf.ones_like(r_logit), r_logit)
        f_loss = bce(tf.zeros_like(f_logit), f_logit)
        return r_loss, f_loss

    def g_loss_fn(f_logit):
        f_loss = bce(tf.ones_like(f_logit), f_logit)
        return f_loss

    return d_loss_fn, g_loss_fn

def get_hinge_v1_losses_fn():
    def d_loss_fn(r_logit, f_logit):
        r_loss = tf.reduce_mean(tf.maximum(1 - r_logit, 0))
        f_loss = tf.reduce_mean(tf.maximum(1 + f_logit, 0))
        return r_loss, f_loss

    def g_loss_fn(f_logit):
        f_loss = tf.reduce_mean(tf.maximum(1 - f_logit, 0))
        return f_loss

    return d_loss_fn, g_loss_fn

def get_hinge_v2_losses_fn():
    def d_loss_fn(r_logit, f_logit):
        r_loss = tf.reduce_mean(tf.maximum(1 - r_logit, 0))
        f_loss = tf.reduce_mean(tf.maximum(1 + f_logit, 0))
        return r_loss, f_loss

    def g_loss_fn(f_logit):
        f_loss = tf.reduce_mean(- f_logit)
        return f_loss

    return d_loss_fn, g_loss_fn

def get_lsgan_losses_fn():
    mse = tf.losses.MeanSquaredError()
```

```

def d_loss_fn(r_logit, f_logit):
    r_loss = mse(tf.ones_like(r_logit), r_logit)
    f_loss = mse(tf.zeros_like(f_logit), f_logit)
    return r_loss, f_loss

def g_loss_fn(f_logit):
    f_loss = mse(tf.ones_like(f_logit), f_logit)
    return f_loss

return d_loss_fn, g_loss_fn

def get_wgan_losses_fn():
    def d_loss_fn(r_logit, f_logit):
        r_loss = - tf.reduce_mean(r_logit)
        f_loss = tf.reduce_mean(f_logit)
        return r_loss, f_loss

    def g_loss_fn(f_logit):
        f_loss = - tf.reduce_mean(f_logit)
        return f_loss

    return d_loss_fn, g_loss_fn

def get_adversarial_losses_fn(mode):
    if mode == 'gan':
        return get_gan_losses_fn()
    elif mode == 'hinge_v1':
        return get_hinge_v1_losses_fn()
    elif mode == 'hinge_v2':
        return get_hinge_v2_losses_fn()
    elif mode == 'lsgan':
        return get_lsgan_losses_fn()
    elif mode == 'wgan':
        return get_wgan_losses_fn()

def gradient_penalty(f, real, fake, mode):
    def _gradient_penalty(f, real, fake=None):
        def _interpolate(a, b=None):
            if b is None: # interpolation in DRAGAN
                beta = tf.random.uniform(shape=tf.shape(a), minval=0., maxval=1.)
                b = a + 0.5 * tf.math.reduce_std(a) * beta
            shape = [tf.shape(a)[0]] + [1] * (a.shape.ndims - 1)
            alpha = tf.random.uniform(shape=shape, minval=0., maxval=1.)
            inter = a + alpha * (b - a)
            inter.set_shape(a.shape)
            return inter

        x = _interpolate(real, fake)
        with tf.GradientTape() as t:
            t.watch(x)
            pred = f(x)
            grad = t.gradient(pred, x)
            norm = tf.norm(tf.reshape(grad, [tf.shape(grad)[0], -1]), axis=1)
            gp = tf.reduce_mean((norm - 1.)**2)

        return gp

    if mode == 'none':
        gp = tf.constant(0, dtype=real.dtype)
    elif mode == 'dragan':
        gp = _gradient_penalty(f, real)
    elif mode == 'wgan-gp':
        gp = _gradient_penalty(f, real, fake)

    return gp

```

```

In [36]: # adversarial_loss_functions
d_loss_fn, g_loss_fn = get_adversarial_losses_fn(ADVERSARIAL_LOSS_MODE)

G_optimizer = keras.optimizers.Adam(learning_rate=LR, beta_1=BETA_1)
D_optimizer = keras.optimizers.Adam(learning_rate=LR, beta_1=BETA_1)

```

Training step

```

In [37]: @tf.function
def train_G():
    with tf.GradientTape() as t:
        z = tf.random.normal(shape=(BATCH_SIZE, 1, 1, Z_DIM))
        x_fake = G(z, training=True)
        x_fake_d_logit = D(x_fake, training=True)
        G_loss = g_loss_fn(x_fake_d_logit)

    G_grad = t.gradient(G_loss, G.trainable_variables)

```

```

G_optimizer.apply_gradients(zip(G_grad, G.trainable_variables))

return {'g_loss': G_loss}

@tf.function
def train_D(x_real):
    with tf.GradientTape() as t:
        z = tf.random.normal(shape=(BATCH_SIZE, 1, 1, Z_DIM))
        x_fake = G(z, training=True)

        x_real_d_logit = D(x_real, training=True)
        x_fake_d_logit = D(x_fake, training=True)

        x_real_d_loss, x_fake_d_loss = d_loss_fn(x_real_d_logit, x_fake_d_logit)
        gp = gradient_penalty(functools.partial(D, training=True), x_real, x_fake, mode=GRADIENT_PENALTY_MODE)

        D_loss = (x_real_d_loss + x_fake_d_loss) + gp * GRADIENT_PENALTY_WEIGHT

    D_grad = t.gradient(D_loss, D.trainable_variables)
    D_optimizer.apply_gradients(zip(D_grad, D.trainable_variables))

    return {'d_loss': x_real_d_loss + x_fake_d_loss, 'gp': gp}

@tf.function
def sample(z):
    return G(z, training=False)

```

Checkpoint:

In [38]:

```

class Checkpoint:
    """Enhanced "tf.train.Checkpoint"."""

    def __init__(self,
                 checkpoint_kwargs, # for "tf.train.Checkpoint"
                 directory, # for "tf.train.CheckpointManager"
                 max_to_keep=5,
                 keep_checkpoint_every_n_hours=None):
        self.checkpoint = tf.train.Checkpoint(**checkpoint_kwargs)
        self.manager = tf.train.CheckpointManager(self.checkpoint, directory, max_to_keep, keep_checkpoint_every_n_ho

    def restore(self, save_path=None):
        save_path = self.manager.latest_checkpoint if save_path is None else save_path
        return self.checkpoint.restore(save_path)

    def save(self, file_prefix_or_checkpoint_number=None, session=None):
        if isinstance(file_prefix_or_checkpoint_number, str):
            return self.checkpoint.save(file_prefix_or_checkpoint_number, session=session)
        else:
            return self.manager.save(checkpoint_number=file_prefix_or_checkpoint_number)

    def __getattr__(self, attr):
        if hasattr(self.checkpoint, attr):
            return getattr(self.checkpoint, attr)
        elif hasattr(self.manager, attr):
            return getattr(self.manager, attr)
        else:
            self.__getattribute__(attr) # this will raise an exception

    def summary(name_data_dict, step=None, types=['mean', 'std', 'max', 'min', 'sparsity', 'histogram'],
               histogram_buckets=None, name='summary'):
        """Summary.

        Examples
        -----
        >>> summary({'a': data_a, 'b': data_b})

        """
        def _summary(name, data):
            if data.shape == ():
                tf.summary.scalar(name, data, step=step)
            else:
                if 'mean' in types:
                    tf.summary.scalar(name + '-mean', tf.math.reduce_mean(data), step=step)
                if 'std' in types:
                    tf.summary.scalar(name + '-std', tf.math.reduce_std(data), step=step)
                if 'max' in types:
                    tf.summary.scalar(name + '-max', tf.math.reduce_max(data), step=step)
                if 'min' in types:
                    tf.summary.scalar(name + '-min', tf.math.reduce_min(data), step=step)
                if 'sparsity' in types:
                    tf.summary.scalar(name + '-sparsity', tf.math.zero_fraction(data), step=step)
                if 'histogram' in types:
                    tf.summary.histogram(name, data, step=step, buckets=histogram_buckets)

        with tf.name_scope(name):

```



```

for name, data in name_data_dict.items():
    _summary(name, data)

```

Image functions:

In [39]:

```

def _check(images, dtypes, min_value=-np.inf, max_value=np.inf):
    # check type
    assert isinstance(images, np.ndarray), '`images` should be np.ndarray!'

    # check dtype
    dtypes = dtypes if isinstance(dtypes, (list, tuple)) else [dtypes]
    assert images.dtype in dtypes, 'dtype of `images` should be one of %s!' % dtypes

    # check nan and inf
    assert np.all(np.isfinite(images)), '`images` contains NaN or Inf!'

    # check value
    if min_value not in [None, -np.inf]:
        l = '[' + str(min_value)
    else:
        l = '(-inf'
        min_value = -np.inf
    if max_value not in [None, np.inf]:
        r = str(max_value) + ']'
    else:
        r = 'inf)'
    max_value = np.inf
    assert np.min(images) >= min_value and np.max(images) <= max_value, \
        '`images` should be in the range of %s!' % (l + ',' + r)

def to_range(images, min_value=0.0, max_value=1.0, dtype=None):
    """Transform images from [-1.0, 1.0] to [min_value, max_value] of dtype."""
    _check(images, [np.float32, np.float64], -1.0, 1.0)
    dtype = dtype if dtype else images.dtype
    return ((images + 1.) / 2. * (max_value - min_value) + min_value).astype(dtype)

def float2im(images):
    """Transform images from [0, 1.0] to [-1.0, 1.0]."""
    _check(images, [np.float32, np.float64], 0.0, 1.0)
    return images * 2 - 1.0

def float2uint(images):
    """Transform images from [0, 1.0] to uint8."""
    _check(images, [np.float32, np.float64], -0.0, 1.0)
    return (images * 255).astype(np.uint8)

def im2uint(images):
    """Transform images from [-1.0, 1.0] to uint8."""
    return to_range(images, 0, 255, np.uint8)

def im2float(images):
    """Transform images from [-1.0, 1.0] to [0.0, 1.0]."""
    return to_range(images, 0.0, 1.0)

def uint2im(images):
    """Transform images from uint8 to [-1.0, 1.0] of float64."""
    _check(images, np.uint8)
    return images / 127.5 - 1.0

def uint2float(images):
    """Transform images from uint8 to [0.0, 1.0] of float64."""
    _check(images, np.uint8)
    return images / 255.0

def cv2im(images):
    """Transform opencv images to [-1.0, 1.0]."""
    images = uint2im(images)
    return images[..., :-1]

def im2cv(images):
    """Transform images from [-1.0, 1.0] to opencv images."""
    images = im2uint(images)
    return images[..., :-1]

```

In [40]:

```

def imread(path, as_gray=False, **kwargs):
    """Return a float64 image in [-1.0, 1.0]."""
    image = iio.imread(path, as_gray, **kwargs)
    if image.dtype == np.uint8:
        image = image / 127.5 - 1
    elif image.dtype == np.uint16:
        image = image / 32767.5 - 1
    elif image.dtype in [np.float32, np.float64]:
        image = image * 2 - 1.0
    else:

```

```

        raise Exception("Inavailable image dtype: %s!" % image.dtype)
    return image

def imwrite(image, path, quality=95, **plugin_args):
    """Save a [-1.0, 1.0] image."""
    iio.imwrite(path, im2uint(image), quality=quality, **plugin_args)

def imshow(image):
    """Show a [-1.0, 1.0] image."""
    iio.imshow(im2uint(image))

def immerge(images, n_rows=None, n_cols=None, padding=0, pad_value=0):
    """Merge images to an image with (n_rows * h) * (n_cols * w).

    Parameters
    -----
    images : numpy.array or object which can be converted to numpy.array
            Images in shape of N * H * W(* C=1 or 3).

    """
    images = np.array(images)
    n = images.shape[0]
    if n_rows:
        n_rows = max(min(n_rows, n), 1)
        n_cols = int(n - 0.5) // n_rows + 1
    elif n_cols:
        n_cols = max(min(n_cols, n), 1)
        n_rows = int(n - 0.5) // n_cols + 1
    else:
        n_rows = int(n ** 0.5)
        n_cols = int(n - 0.5) // n_rows + 1

    h, w = images.shape[1], images.shape[2]
    shape = (h * n_rows + padding * (n_rows - 1),
            w * n_cols + padding * (n_cols - 1))
    if images.ndim == 4:
        shape += (images.shape[3],)
    img = np.full(shape, pad_value, dtype=images.dtype)

    for idx, image in enumerate(images):
        i = idx % n_cols
        j = idx // n_cols
        img[j * (h + padding):j * (h + padding) + h,
           i * (w + padding):i * (w + padding) + w, ...] = image

    return img

```

Run

In [17]:

```

# epoch counter
ep_cnt = tf.Variable(initial_value=0, trainable=False, dtype=tf.int64)

# checkpoint
checkpoint = Checkpoint(dict(G=G, D=D, G_optimizer=G_optimizer, D_optimizer=D_optimizer, ep_cnt=ep_cnt),
                        os.path.join(output_dir, 'checkpoints'), max_to_keep=5)
try: # restore checkpoint including the epoch counter
    checkpoint.restore().assert_existing_objects_matched()
except Exception as e:
    print(e)

# summary
train_summary_writer = tf.summary.create_file_writer(os.path.join(output_dir, 'summaries', 'train'))

# sample
sample_dir = os.path.join(output_dir, 'samples_training')
mkdir(sample_dir)

# main loop
z = tf.random.normal((100, 1, 1, Z_DIM)) # a fixed noise for sampling
with train_summary_writer.as_default():
    for ep in tqdm.trange(EPOCHS, desc='Epoch Loop'):
        if ep < ep_cnt:
            continue

        # update epoch counter
        ep_cnt.assign_add(1)

        # train for an epoch
        for x_real in tqdm.tqdm(dataset, desc='Inner Epoch Loop', total=len_dataset, disable=True):
            D_loss_dict = train_D(x_real)
            summary(D_loss_dict, step=D_optimizer.iterations, name='D_losses')

            if D_optimizer.iterations.numpy() % N_D == 0:
                G_loss_dict = train_G()
                summary(G_loss_dict, step=G_optimizer.iterations, name='G_losses')

```

```

# sample
if G_optimizer.iterations.numpy() % PRINT_SAMPLE == 0:
    x_fake = sample(z)
    img = immerge(x_fake, n_rows=10).squeeze()
    imwrite(img, os.path.join(sample_dir, 'iter-%09d.jpg' % G_optimizer.iterations.numpy()))

# save checkpoint
checkpoint.save(ep)

```

Epoch Loop: 100%|██████████| 115000/115000 [01:48<00:00, 1064.79it/s]

Discriminator iterations:

```
In [41]: print(D_optimizer.iterations.numpy())
```

0

Generator iterations:

```
In [42]: print(G_optimizer.iterations.numpy())
```

0

Result images:

```
In [23]: def print_100_single_images(epoch):
images = []
idx_images = []
def show_images(tester):
    for idx, image in enumerate(tester):
        images.append(image)
        idx_images.append(idx)
    return idx_images, images
result_name = 'result_images'+str(epoch)
result_dir = os.path.join(output_dir, result_name)
mkdir(result_dir)

img_sample = sample(z)
tester = np.array(img_sample)
idx_list, image_list = show_images(tester)
for i in idx_images:
    imwrite(image_list[i], os.path.join(result_dir, 'idx-%09d.jpg' % i))

```

```
In [24]: print_100_single_images(EPOCHS)
```

This copy of the code was downloaded from Google Colab and opened in Jupyter to provide a better presentation in the pdf.

The code below can be used to clone the needed GitHub repository files. For this code the repository has been uploaded to a Google Drive folder named yolov5-master

```
In [ ]: #!git clone https://github.com/ultralytics/yolov5
```

The program steps into the files location folder on Google drive, install requirements (can be found at <https://github.com/ultralytics/yolov5>) and preform imports.

```
In [1]: %cd /content/drive/MyDrive/yolov5-master
%pip install -qr requirements.txt # install dependencies

import torch
from IPython.display import Image, clear_output # to display images

clear_output()
print(f"Setup complete. Using torch {torch.__version__} ({torch.cuda.get_device_properties(0).name if torch.cuda.is_
```

Setup complete. Using torch 1.8.1+cu101 (Tesla K80)

Test that the code runs:

```
In [ ]: !python detect.py --weights yolov5s.pt --img 640 --conf 0.25 --source data/images/
Image(filename='runs/detect/exp/zidane.jpg', width=600)
```

```
Namespace(agnostic_nms=False, augment=False, classes=None, conf_thres=0.25, device='', exist_ok=False, hide_conf=False, hide_labels=False, img_size=640, iou_thres=0.45, line_thickness=3, max_det=1000, name='exp', nosave=False, project='runs/detect', save_conf=False, save_crop=False, save_txt=False, source='data/images/', update=False, view_img=False, weights=['yolov5s.pt'])
YOLOv5 2021-5-26 torch 1.8.1+cu101 CUDA:0 (Tesla K80, 11441.1875MB)
```

Fusing layers...

Model Summary: 224 layers, 7266973 parameters, 0 gradients

image 1/2 /content/drive/My Drive/yolov5-master/data/images/bus.jpg: 640x480 4 persons, 1 bus, 1 fire hydrant, Done. (0.032s)

image 2/2 /content/drive/My Drive/yolov5-master/data/images/zidane.jpg: 384x640 2 persons, 2 ties, Done. (0.030s)

Results saved to runs/detect/exp6

Done. (0.201s)

```
Out [ ]:
```



Tensorboard

```
In [ ]: %load_ext tensorboard
%tensorboard --logdir runs/train
```

Weights & Biases, not used.

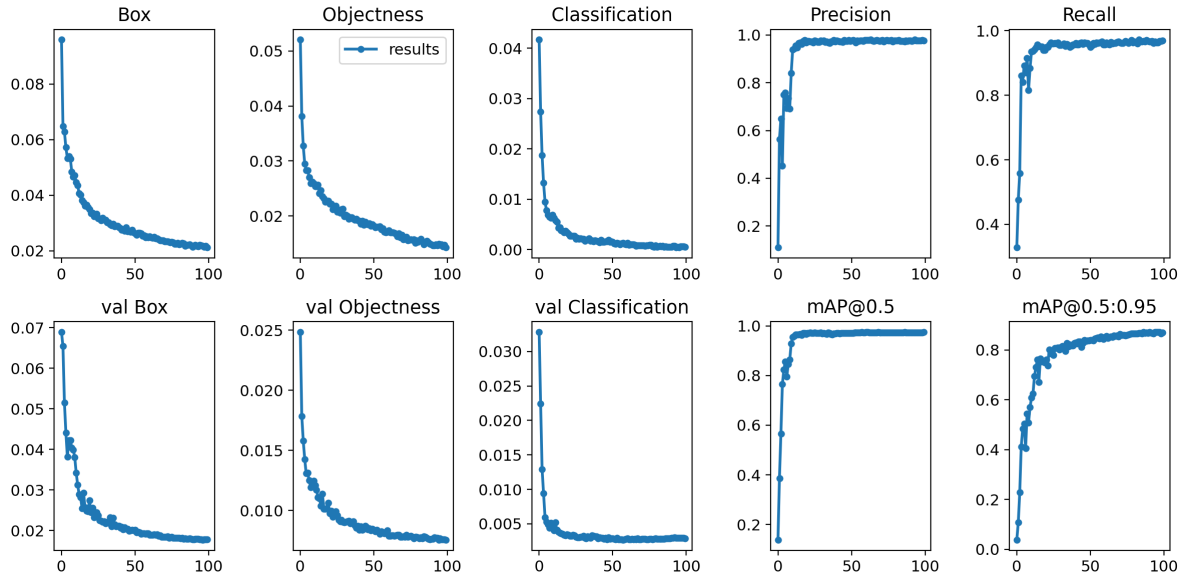
```
In [ ]: #!pip install -q wandb
import wandb
#wandb.login()
```

Training:

```
In [ ]: !python train.py --img 256 --batch 16 --epochs 100 --data data_v2.yaml --cfg yolov5x.yaml --weights 'yolov5x.pt' --name
```

```
In [ ]: from utils.plots import plot_results
plot_results(save_dir='runs/train/yolov5x_100epoch_bs64') # plot all results*.txt as results.png
Image(filename='runs/train/yolov5x_100epoch_bs64/results.png', width=800)
```

Out [ ]:



Testing the YOLOv5x after training:

```
In [ ]: !python detect.py --weights runs/train/yolov5x_100epoch_bs64/weights/best.pt --img 256 --conf 0.25 --source data/image
```

Testing the YOLOv5x after training (with a smaller detection line thickness):

```
In [ ]: !python detect.py --weights runs/train/yolov5x_100epoch_bs64/weights/best.pt --img 256 --conf 0.25 --source data/image
```

Testing the YOLOv5x on WGAN-GP generated images:

```
In [ ]: !python detect.py --weights runs/train/yolov5x_100epoch_bs64/weights/best.pt --img 256 --conf 0.25 --source data/image
```